東京大学教育用計算機センターテキスト

Pascalプログラミング

TURBO Pascalによる基礎と応用［第2版］

永野三郎・長島 忍・吉村 伸

東京大学出版会

「東京大学教育用計算機センターテキスト」について

東京大学教育用計算機センターテキストは，情報処理教育にとって役立つ書を次々と発刊する目的で作られた，シリーズ色の強い企画である．

我々がお手本としてきたのは，「JIS FORTRAN 入門」（森口繁一著）である．この書は短期にしかも独学でプログラミングを習得できるものとして優れた本である．この思想を見習い，従来極めて使いづらかった大型機のマニュアルを，独学でも使えるようにと工夫したのが，本テキストの第一作『計算機センタ利用入門』である．この書は，本センターに昭和 56 年に設置された MEL-COM COSMO 900 II が，TSS の事実上初めての機種だったことから，広範な TSS の使い方を学生が自習できるように立案されたものである．こうした書は従来，ある程度使いこなしたユーザーを対象とするメーカーのマニュアルか，比較的完成度の低い各センターのプリントの形のものしかなかったものをあえてテキスト化したものであるが，比較的評判も良く，こうした指導書に対するある程度の方向を示したものと確信している．

その後，パソコンが広範囲に普及し，ある程度高度な仕事もパソコンで取扱うことが可能となってきた．本センターでも昭和 61 年より，FACOM M-380/M-360 をホストとし，パソコン FM-16 β を端末とする新機種が導入された．この際，パソコンによく整合し，また他の高級言語の基礎としても極めて優れた言語であるとして，当センターが初等情報処理教育用に選んだのが，(Turbo) PASCAL である．この言語はソフトウエア開発の基本である構造化された言語の代表であり，現在システム開発によく使われる C にも似ており，欧米ではもっとも多く教育に使われているにもかかわらず，日本では実用的言語教育のみが強く，あまり普及していないようである．そこで本センターでは，この言語を第二作として取り上げた．本言語は東京大学の教養学部における初等情報処理教育の大部分で採用される予定であり，他大学にも広まっていくことが，情報処理教育の将来のために望ましい姿であると確信している．

本センターでは，今後も，計算機教育にとって望ましい方向を示唆する書を，本センターテキストとして企画していくつもりであり，読者の方々からの適切な御叱責，御助言をいただけたら幸いである．

東京大学教育用計算機センター　業務委員会

ま え が き

電子計算機がこの世に出現して以来，僅かに40年の歳月を経たに過ぎない．しかし，この間に電子・通信・情報技術は目まぐるしいまでに進歩・発達を遂げ，現在では計算機とその利用技術が社会のすみずみにまで浸透して，人間社会全体に大きなインパクトを与えている．実際，我々がそれと意図，ないし認識すらしないうちに，日常生活のさまざまな場面で，コンピュータが活用されており，我々の生活様式，価値観，文化総体に対してすら少なからぬ影響を及ぼしつつある．昭和60年４月からの電気通信事業の自由化を契機として，全国規模，世界規模のコンピュータネットワークの構築も急速に進み，来たるべき情報化社会は次第に身近なものとなりつつある．

コンピュータの効用は，勿論このような日常的生活のレベルにばかりあるのではない．むしろ，コンピュータの出現によって，我々の知的活動領域の幅が格段に拡大されたところに，より大きな意義がある．人類の歴史において，我々の祖先は進歩とともにその活動の一部を道具や機械に代行させ，そうすることによって得られたゆとりを用いて，更に活動領域を拡大してきた．太古において人類は道具の使用を開始し，それによって人間らしい生活の第一歩を踏み出した．18世紀の産業革命は動力機械の発明が契機となっており，これによって人類は激しい肉体労働から解放された．そして現代はコンピュータと電気通信技術の進歩によって，人間の最も高級な活動である情報活動の一部機械化を可能にし，我々の活動領域を急速に拡大しつつある時代であると位置づけることができよう．実際，直径１ミリにも満たない光ファイバーを介して，僅か１秒間に文庫本数十冊分にも相当する量の情報がやりとりされ，数十センチ四方の箱の中で，毎秒数十億回という，地球上の全ての人間が束になって掛ってもかなわないほどのスピードで，計算をこなすスーパー・コンピュータが何十台も日夜フル稼働している．これらはいずれもより高度な情報を求め，新しい知見を求めて，あるいは人類未踏の技術を追求して行われる情報処理活動の一端である．

さらに，このように人間の知的活動を一部代行する機械が出現したことによって，人間の「知」とは何かが改めて問われることとなり，計算機とその利用技術は哲学，言語学，心理学等の人文系諸学問分野とも深い関わりを持つに至り，情報科学，認知科学，知識工学などの新しい学問分野も生れた．記憶，学習，知覚，判断，推論など，人間の知的活動を代行する機械としての人工知能の研究が進むにつれて，人間そのもののより深い理解が求められ，既存の諸学問分野に刺激を及ぼすなど，計算機をめぐる科学はあらゆる学問分野と相互に影響を及ぼし合いつつ，進歩・発展をつづけている．

以上を要するに，計算機は今や，自然科学や科学技術の分野のみならず，人文科学，社会科学など，あらゆる学問分野における共通の道具ないし対象となっており，計算機および情報処理の入門教育は大学における一般教育として必要不可欠のものとなってきている．丁度「読み・書き・そろばん」が，かつて人間社会の中で生きていくために必須の基本素養であったように，また外国語の

修得が国際化社会の中にあっては不可欠の要件であるように，コンピュータを利用した情報の生成，加工，伝達，蓄積，利用等ができることは今後の高度情報化社会においては必須の基本素養となってきているわけである．

東京大学では，昭和62年度から，1，2年生の学生諸君を対象に，電子計算機および情報処理に関する入門教育を，教養学部における一般教育の一環として本格的に実施しはじめることとなった．そのカリキュラム内容については，教養学部の内部で昭和53年以来，検討と準備が進められてきた．その際，以下の諸点が前提として考えられた．

(1)　計算機の基本的性質を理解するためには，プログラミングの実習が不可欠である．

(2)　100人以上の多人数一斉教育となるから，パーソナル・コンピュータの処理能力を最大限に生かした，LANのもとでの教育を基本とする．

(3)　単なる計算機操作法やある特定のプログラミング言語の修得にとどまらず，激しく変化していくであろう計算機や情報処理に関わる科学・技術情報を，学生諸君が将来にわたって学習・理解・利用・評価していく際の基盤となる確かな枠組を与えることを目標とする．

以上の観点から，大学1，2年生を対象とした理想的な情報処理入門教育の具体的内容を考えるならば，次のような項目が挙げられるであろう．

(a)　電子計算機概論(作動原理，基本機能，ソフトおよびハードウエア上の基本構成，データの内部表現等)

(b)　ハードウエア機器の操作法およびシステムソフトウエア機能(OS，DOS，EDITOR，FILE操作等)の利用法

(c)　計算機言語の文法要説(PASCAL，FORTRAN，C，BASIC等のうちのいずれか一つ)

(d)　基本的算法(繰返し計算とその収束性，連続量の離散化と誤差，ソーティングのアルゴリズム，グラフィクスの初歩等)

(e)　応用情報処理概論(図形処理，信号処理，数式処理，パターン認識，人工知能等)

(f)　情報処理の科学概論(計算可能性，問題処理の複雑さ，NP完全性等の理論)

ただし，初等中等教育において計算機教育が正規の授業としては行われていない現状で，毎週2時間，1個学期(15週)で完了という程度の入門教育とすれば，(a)～(d)を講義と演習を通じてバランス良く体得してもらうだけで手一杯となりそうである．やがて初等中等教育に(a)～(c)などが取り入れられるようになれば，大学教養課程における教育も(d)～(f)へとシフトさせていくことになろう．

本書は，前記(1)～(3)の前提の上に立って，大学初年度の理科系学生に対する教科書として，上述の(a)～(d)の部分を1冊にまとめたものである．PASCALプログラミングに関する優れた入門書・教科書が既に多数出版されている中で，本書を上梓することに躊躇しなかったわけではない．しかし，東京大学教育用計算機センター業務委員会から強いお勧めをいただいたこと，および，前

記(a)〜(d)を1冊にまとめることができれば，入門教育を受ける学生諸君は，従来のように操作書・文法書・その他の教科書等を何冊も参照する必要もなく，より手軽にかつ能率的に学習ができるのではないかと考えたこと，が動機となって敢えて執筆に踏み切った次第である．

本書の第I部(1〜3章)においては，計算機に関する一切の予備知識を前提とせずにPascal言語によるプログラム作成とその実行に必要な入門事項を述べた．第II部の4，5，6章はたがいにほぼ独立のものと考えてよく，授業時間数や計算機実習に費やすことのできる時間数に合わせて適宜取捨選択が可能である．また，本書は一応理科系学生用教科書として構想されているが，第4章の一部を統計処理や言語処理の課題に置き換えることによって，文科系学生用教科書としても無理なく利用できるよう配慮したつもりである．

なお本書の執筆にあたってはマイクロソフトウエアアソシエイツ社のTURBO Pascalシステムを使用し，そのプログラミングマニュアルを参照した．執筆分担は，1，2章が永野三郎，3，6章および付録が長島忍，4，5章が吉村伸である．また例題のプログラム作成には東京大学大学院総合文化研究科広域科学専攻の松原敦君，牧野淳一郎君の協力を得た．執筆分担者は互いに連絡を取り，まとまりを良くするよう努めたが，必ずしも充分とは言えない面もあろうかと惧れている．大方の御批判，御叱責を得て改良していくことができれば幸いである．

本書作成の機会を与えてくださった東京大学教育用計算機センター業務委員会，なかんずくPASCALの採用と本書の執筆を強力に勧めてくださった岡部洋一(前)センター主任および吉澤昭宣(元)業務委員長に感謝したい．また本書に盛り込まれた情報処理入門教育のカリキュラムを長時間の検討の末まとめ上げたうえ，本書の第I部について初校に目を通し，有益な助言をくださった東京大学教養学部「情報・図学教室」の同僚諸兄，ならびに遅れがちな筆者らの原稿を辛抱強く，寛容に待ち続けてくださった上，何かと奔走してくださった東京大学出版会の小池美樹彦氏に感謝の意を表したい．

1987年3月

執筆者代表

永野　三郎

題(b)一覧表(1画)にまとめることができれば，入門教育を受ける学生諸君は，従来のように[illegible]書と文法書，その他の教科書類を同時に参照する必要もなく，より手軽にかつ能率的に学習ができるのではないかと考えたこと，が動機となって[illegible]

本書の第1部(1～3章)においては，計算機に関する一切の予備知識を前提とせずにPascal言語による[illegible]一部を[illegible]利用できるよう配慮したものである。[illegible]

[illegible] TURBO Pascal [illegible]を使用し，[illegible]

[illegible]

本書[illegible]PASCAL の採用[illegible]

[illegible]の意を表したい。

[illegible]

改訂にあたって

本書が刊行されてから5年が経過した．このテキストは，東京大学教養学部において，大学1年生を対象とした情報処理基礎教育が正規の授業科目として開設されるのに合わせて刊行したものであった．プログラミング演習を含む情報処理の基礎教育においては，プログラムの作成・編集・実行の各段階で，使用する計算機システムのハードウェアおよびソフトウェアに依存する部分も少なくない．本書においては，近年機能向上が著しく，広く普及しているパーソナルコンピュータを使用し，かつプログラミング言語としてはMS-DOS上のTURBO Pascalを使用するものとして，パソコンの操作法，TURBO Pascalシステムの利用法，ならびにプログラミング言語の文法，基礎的なアルゴリズム等のすべてを1冊の本にまとめようとしたものであった．

周知のごとく，パソコンのオペレーティングシステムや，その上の言語処理システムの機能とその利用法は年々変化していく．多くの場合に新しいバージョンは古いバージョンと両立し得る形に作られるから，ソフトウェア側の変更の度に本書の内容を書き改めることは必ずしも必要不可欠ではない．しかし，一つは，TURBO Pascalが提供する機能とその利用法に関するこの5年間の変化がきわめて大きいこと，いま一つには，5年間の教育実施経験と，旧著を使用された方々の御助言をできる限り取り入れたいと考えたことから，大幅な改訂を行うこととした．改訂の主要な点は以下のとおりである．

(1)　TURBO Pascalの操作法に関する説明(第2章)はVer. 5を標準とした．

(2)　プログラミング入門教育の基礎となる第3章は，ほとんど全面的に書き改めるとともに，各節ごとに練習問題を付け加えて自習の助けとした．

(3)　東京大学教養学部では，文科系・理科系を問わず1年次にプログラミング演習を含む情報処理基礎教育を行い，理工系学部学科進学予定者に対しては，2年次に3次元図形処理(ソリッドモデリング)を中心とするコンピュータグラフィックス教育を施している．本書によって，本格的なコンピュータグラフィックス教育のテキストを兼ねるには無理があるので，本書の内容は1年次の使用のみに限定し，第6章のグラフィックスの部分はGraphユニットの紹介と平面図形の取り扱いに限定した．

(4)　TURBO Pascal Ver. 5以降では，Graphユニットの他にも，Crtユニットや Dosユニットなど，使い勝手のよい機能が提供されている．これらについても第6章で簡単に紹介するとともに，付録に要点をまとめて示した．

この版の序も私が書かせていただくことになったが，改訂作業の殆どすべては共著者の長島忍氏が引受られたものであり，同氏の労を謝したい．

1992年2月

永　野　　三　郎

改訂にあたって

本書が刊行されてから5年が経過した。このテキストは，東京大学教養学部において，大学1年生を対象とした情報処理基礎教育が正規の授業科目として開講されるのに合わせて刊行したものであった。プログラミング演習を含む情報処理の基礎教育においては，プログラムの作成・編集・実行の各段階で，使用する計算機システムのハードウェアおよびソフトウェアに依存する部分も少なくない。本書においては，近年機能向上が著しく，広く普及しているパーソナルコンピュータを使用し，かつプログラミング言語としてはMS-DOS上でTURBO Pascalを使用するものとして，パソコンの操作法，TURBO Pascalシステムの利用法，ならびにプログラミング言語の文法・基礎的なテクニック等のすべてを1冊の本にまとめようとしたものであった。

周知のごとく，パソコンのオペレーティングシステムや，その上の言語処理システムの機能とその利用法は年々変化している。多くの場合は新しいバージョンは旧バージョンと両立し得る形に作られるから，ソフトウェア側の変更の度に本書の内容を書き改めることは必ずしも必要不可欠ではない。しかし，一つは，TURBO Pascal の提供する機能とその利用法に関するこの5年間の変化があまりにも大きいこと，いま一つには，5年間の教育実施経験と，旧著を使用された方々の御助言をできる限り取り入れたいと考えたことから，大幅な改訂を行うこととした。改訂の主要な点は以下のとおりである。

(1) TURBO Pascalの操作法に関する説明（第2章）はVer. 5を標準とした。

(2) プログラミング入門教育の基礎となる第3章は，ほとんど全面的に書き改めるとともに，各節ごとに練習問題を付け加えて自習の助けとした。

(3) 東京大学教養学部では，文科系・理科系を問わず1年次にプログラミング演習を含む情報処理基礎を課しており，理工系学部学科進学予定者に対しては，2年次に3次元図形処理（ソリッドモデリング）を中心とするコンピュータグラフィックス教育を実施している。本書によって，本格的なコンピュータグラフィックス教育のテキストを兼ねるには無理があるので，本書の内容は1年次の教育のみに限定し，第8章のグラフィックスの部分はGraphユニットの紹介と平面図形の取り扱いに限定した。

(4) TURBO Pascal Ver. 5以降では，Graphユニットの他にも，Crtユニット，Dosユニットなど，使い勝手のよい機能が提供されている。これらについても第9章で簡単に紹介するとともに，付録に要点をまとめておいた。

この版の改訂も私が書かせていただくことになったが，[illegible]の労を煩わせたものであり，同氏の労を謝したい。

[illegible]

[illegible]

目　　次

第II部　応　用　編

第I部　基　礎　編

第I部　基　礎　編

第1章
プログラミング入門

1.1　電子計算機の概要

計算機に対する一連の作業指示をプログラムという．また，これらプログラムの集まりをその計算機のソフトウエアと総称する．これに対して，物理的な装置としての計算機システムを構成する電子部品や機械装置をハードウエアと呼んでいる．本書のほとんどすべては，計算機を利用する立場から，それに必要なプログラミングに関わる入門書であるが，ハードウエアの基本構成をわきまえておくことは，以後の計算機利用に際し有益であろう．この節では計算機のハードウエアおよびソフトウエア上の基本構成について概説する．

(a)　ハードウエアの基本構成

一口に計算機と言っても，現在では1個数千円のマイクロプロセッサから，1台数十億円もする汎用大型計算機まで，価格や性能(処理速度等)において10^6倍もの差があるほどに，規模や性能の異なる計算機群がある．

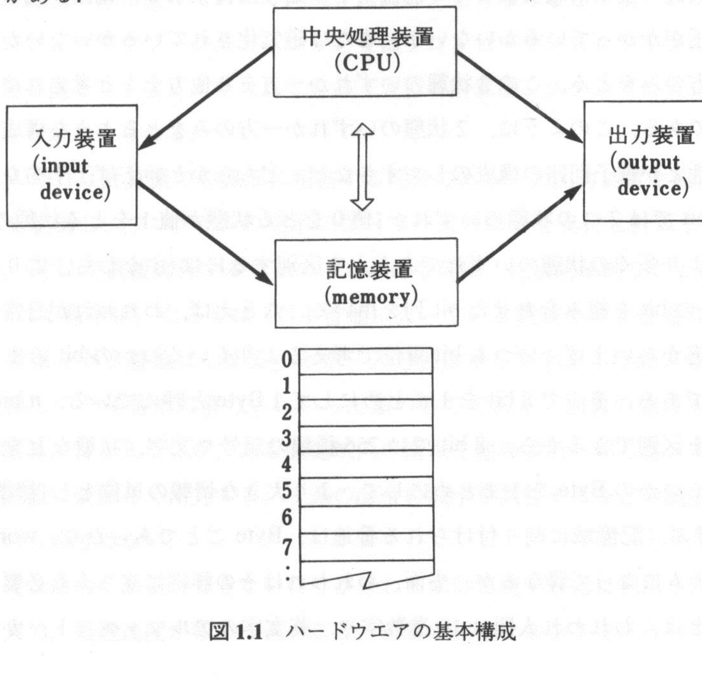


図 1.1　ハードウエアの基本構成

しかし，われわれが通常身近で利用する計算機のほとんどは，図1.1のようなハードウエア構成になっていると考えて差し支えない．すなわち，計算機システムの多くは，入力装置，出力装置，中央処理装置(Central Processing Unit，以下CPUと略記する)および記憶装置の4つの部分からなっている．

入力装置は，プログラムやプログラムの実行に必要なデータ等の情報を計算機に与えるためのもので，われわれ人間が直接計算機と対話する場合には，通常，タイプライタと類似したキーボードが用いられる．より高度な利用では，後述の磁気ディスクや磁気テープ，文字や図面・画像の読み取り装置，音声入力装置，TVカメラ，各種センサ等，さまざまなものが入力装置として用いられる．

出力装置は逆に，計算機が生成した情報を外部に伝えるためのもので，われわれが当面利用する出力装置としては，TV画面に類似したCRT(Cathode Ray Tube)ディスプレイ装置がある．その他，各種のプリンタやプロッタ，保存用記憶媒体等が用いられる．

入力情報を保持し，各種の処理の経過や中間結果を書き込んでおくために，計算機には記憶装置が不可欠である．記憶装置自身は，単に，一連の番号(番地)が割り振られた記憶域からなっており，それぞれに番地を付された記憶域の内容自体は，われわれが黒板にチョークで文字や数値を書いたり消したりするように，必要に応じてCPUからの指示で情報を書き込んだり消し去ったりすることができる．

記憶装置や中央処理装置など計算機の内部では，すべての情報は2進数字の列で表現されていると考えることができる．すなわち，計算機内部における情報の最小単位は，数学的には0かまたは1のいずれかの値をとる2進数字1桁であり，これをbit(binary digit)と呼ぶ．物理的には，記憶装置や中央処理装置の基本構成要素である記憶素子や論理演算素子は，電流が流れているかいないか，あるいは電圧がかかっているかいないか，または磁気化されているかいないか，など2つの状態のいずれか一方のみをとる．この2状態のいずれか一方を0他方を1と考えれば，2進数字1桁に相当するわけである．このように，2状態のいずれか一方のみをとるような構成となっているのは，装置の信頼性とか電子回路の構成のしやすさなど，どちらかといえば工学的な理由による．

ところで，1 bitでは2つの状態のいずれか(値0をとる状態か値1をとる状態のいずれか)しか区別できない．より多くの状態のいずれであるかを区別するには(すなわち，より複雑な情報を扱うには)2つ以上のbitを組み合わせたbit列を用いる．たとえば，われわれが日常多用する文字や数字などとの対応からいえば，いつもbit単位で考えるよりもいくつかのbitのまとまりを扱う方が実際的で便利である．そこで8 bitを1まとめにして1 Byteと呼んでいる．n bitでは2^n種類の状態または情報を区別できるから，8 bitでは256種類の記号や文字，状態などを区別することができる．更にいくつかのByteを1まとめにして，より大きな情報の単位とし(標準2〜16 Byte)，これをwordと呼ぶ．記憶域に割り付けられる番地は，Byteごとであったり，wordごとであったり，個々のシステムによって異なるが，当面，われわれはその詳細に立ち入る必要はないであろう．ここで重要なことは，われわれ人間は10進数字や，英文字のアルファベット，カタカナ，ひらか

な，漢字，各種の数学記号などを用いるが，計算機の内部ではこれらのすべてが，何種類かの互いに異なる法則に従って2進数字の列に変換されて，記憶され処理されているという事実である．たとえば，計算機の内部で，ある1Byte分のbit列の中味が

0 1 0 0 0 0 0 1

であったとして，これがJIS (Japan Industrial Standard)に基づく符号化法による文字であるとすれば，英文字の「A」を表わし，整数であるとすれば数値「65」を表わす．実数であるとすれば，小数点の位置をどこに置くと考えるかなど，別の取り決めが要る．したがって，ある特定のビット列が何を表わしているか，あるいは何を表わしていると解釈するかによって，そのビット列が表わす情報の実質的な意味内容が異なってくるのである．幸いプログラムを作るわれわれ人間は，上記の事実さえわきまえていれば，今後，個々のビット列そのものや符号化法に関してその詳細に立ち入ることは，特別の場合を除いて必要でない．そうした面倒なことは当然計算機の側で処置すべきことだからである．

記憶装置は通常何段階かの階層状の構成となっている．いわゆる主記憶装置は，中央処理装置と直結していて，あるプログラムが実行されるときなど，両者の間で頻繁に情報のやりとりが行なわれる．したがってCPUから高速にアクセスできる比較的小容量の記憶装置である．もっとも，近年のLSI，超LSI (VLSI)技術の進歩によって，大型の計算機システムでは数十MByte (1 kByteは1024 Byte，1 MByteは1024 kByte)を越える主記憶容量を持つものも少なくない．

主記憶装置のほかに，大部分の計算機では，2次記憶装置とか補助記憶装置と呼ばれる記憶装置を持っている．プログラムやデータを長時間保存したり，大きな記憶域を必要とするプログラムの実行に際して主記憶のみでは容量が不足する場合に，当面必要な部分だけを主記憶上に置き，そうでない部分は補助記憶の方に置いておいて，適宜入れ替えを行なうなどの使い方が一般的である．パーソナル・コンピュータやワークステーションではフロッピーディスクや小型のハード(固定)ディスクが補助記憶装置として広く用いられている．大型の計算機システムでは大容量の磁気ディスクや磁気テープなどが用いられる．

中央処理装置(CPU)は計算機の心臓部に相当する部分である．入出力装置や記憶装置とCPUの間の情報の流れを制御し，プログラムの指示に従って一連の命令を実行し計算を行なうのもCPUである．しかし，CPUが直接実行することのできる命令群は，比較的単純なことばかりである．

CPUで行なわれる命令は，たとえば「N 1番地に入っている数値をN 2番地に入っている数値に足しあわせ，結果をN 3番地にしまえ」とか「入力装置から文字を1字読み込み，それを0と1の数字列に変換してN 4番地に置け」といった類のものである．加算の他にも，減算，乗算，除算などの四則演算が行なえるほか，ある記憶域内の情報を他の場所へ移し替える，2進数字の列を通常の文字や数値に変換して出力する，一連の命令の順序を調整するなどの機能も持っている．しかしそれ以上のものではない．

このように単純な命令しか実行できないにもかかわらず計算機が有効に利用されているのは，その記憶容量が膨大で，処理速度が極めて速いこと，計算が正確であること，そしてまたわれわれが

作るプログラムによっては，このような単純な命令の組み合わせによっても非常に複雑な計算や処理ができること，などによる．

なお，現在の計算機では，プログラム自身も記憶域の中に保持される．CPUは記憶域の中にあるプログラムを順次読み出しては，その指示に従って一連の計算(情報処理)を行なう．したがって，記憶域はプログラムの保持と，そのプログラムの実行に必要なデータの記憶の両方の目的に利用される．

(b) ソフトウエアの基本構成

計算機は，ハードウエアとしての電子機械装置そのものだけでは，何の機能も果たすことができない．計算機をして有効に目的に沿った仕事を実行させるには，そのためのソフトウエアが不可欠である．ソフトウエアはプログラムの集まりであって目に見えるものではないが，計算機システムの使いやすさや効率の良さなどもソフトウエアに依存するところが大きい．

計算機のハードウエアシステムが大型化・高速化するにつれて，ソフトウエアもまた，その時代の要請に応えるべく著しい進歩・発展を遂げてきた．もっとも，ハードウエアの進歩に比べてソフトウエアの進歩は遅れていると言われる．たとえばハードウエアとしての記憶素子や論理素子が過去40年の間に，真空管→半導体→IC→LSI→VLSI，と進化するにつれて，各時代の先端的コンピュータの最高計算速度は，6～7年ごとに10倍程度ずつ速くなってきており，コスト・パフォーマンスや信頼性も飛躍的に改善されているのに対し，複雑化し巨大化したソフトウエアの信頼性や生産性はそれほど急速に上昇しておらず，ソフトウエアの開発が需要に応じ切れなくなるのではないか，との危機感も根強く存在している．ソフトウエアの生産性を高めるためのさまざまな研究も各方面で進められているが，元来ソフトウエアの開発やプログラムの作成は人間の知的活動に頼らざるをえない面も多く，大幅な自動化や省力化が急速に実現されるとは考え難い．しかし，幸い近年では，ハードおよびソフトの両面で手軽に計算機が利用できる環境が整備されてきており，計算機のユーザ数は急速に増大しつつある．適切なプログラミング教育が広範囲に施され，誰もがわかりやすく質の良いプログラムを書くことができるようになれば，社会全体としてのソフトウエアの生産能力は今後益々上昇していくものと期待される．

さて，計算機ユーザの立場から見る場合には，ソフトウエアの基本構成を図1.2のように分類して考えるとわかりやすいであろう．

オペレーティングシステム(OS)は，計算機を有効に作動させるためにシステム全体の制御・管理・監視等を行なうプログラムで，ハードウエアと最も密着した部分である．大型の汎用計算機では多数のユーザがこれを共用するのが普通であるから，多種多様なユーザからの要求を効率良く処理するためのジョブスケジューリング，記憶領域の割り付けと保護，補助記憶装置との連繋，他のシステムや入出力装置・端末等との間の通信制御，緊急度の高いタスクの割り込み処理，異常事態処理等々，OSの仕事は極めて多岐にわたる．

言語プロセッサは，われわれ人間が日常用いる言葉に比較的近いプログラミング用の言語(高レ

- ソフトウエア
  - システム・ソフトウエア
    - オペレーティング・システム
    - 各種言語プロセッサ
    - サービス／ユーティリティ・プログラム
    - ソフトウエア・ツール
    - ・・・・・・・
  - 数学的ソフトウエア
    - 数値計算プログラム
    - 確率・統計計算プログラム
    - 図形処理プログラム
    - ・・・・・・・
  - 応用ソフトウエア
    - ワード・プロセッサ
    - 事務計算プログラム
    - データベース
    - エクスパート・システム
    - 通信支援システム
    - 教育支援システム
    - ・・・・・・・
  - ユーザ開発ソフトウエア
    - ユーザ・プログラム群
    - ・・・・・・・

図 1.2　ソフトウエアの基本構成

ベル言語)でプログラムを書き計算機に入力したとき，CPU がプログラムの指示どおり直接実行できるように，高レベル言語を2進数字の列(機械語)に翻訳・変換するためのソフトウエアである．高レベル言語としてはPascalの他に，汎用性の高いものでは，C, FORTRAN, BASIC, COBOL, LISP, PROLOG 等多種のものがある．

Pascalなどの高レベル言語でプログラムを書いた場合，一つの文(statement)には，CPUで直接実行できる簡単な命令(instruction)よりはるかに複雑な処理内容を含むこととなるから，その内容を簡単な命令群の組み合わせに翻訳しなおす必要がある．計算機が直接理解することのできる言語は低レベル言語と呼ばれる．1例を挙げれば

```
MOV   AX, N 1
ADD   AX, N 2
MOV   N 3 , AX
```

によって「記憶域内のN1番地に入っている数値をN2番地に入っている数値に足しあわせ，結果をN3番地にしまえ」という意味に用いることができる(ここでAXはCPU内に設けられたレジスタと呼ばれる置数器を指す)．このような表記はアセンブリー言語と呼ばれている．これをCPUに実行させるには，やはり2進数字の列に翻訳する必要がある．これは各CPUについてとりきめがあり，それに従って2進数字の列への符号化が行なわれる．それが機械語である．

高レベル言語を機械語に翻訳するプログラムはコンパイラ(compiler)と呼ばれる．コンパイラはしたがって一風変わったプログラムであって，(高レベル言語で書かれた)プログラムを入力とし，出力も(機械語に翻訳された)プログラムである．混乱を避けるために前者をソースプログラム，後者をオブジェクトプログラムと称して区別する．

サービス/ユーティリティ・プログラムには，プログラマがソースプログラムやデータを作成する際に，能率的な作業を支援するためのテキスト・エディタや，複数のオブジェクトプログラムを1本に結合するリンケージ・エディタ，メモリーにオブジェクトコードをロードするローダなどが含まれる．このうちわれわれに最も関係が深いのはテキストエディタであって，以後単にエディタといえばテキストエディタを指すものとしよう．

さて，高レベル言語で書かれたプログラムがあったとして，このプログラムを計算機に実行させるには，次のような手順を踏むことになる．

(1) まずキーボードからソースプログラムを計算機に入力する．その際，エディタを用いてプログラムの編集や訂正を行なうことができる．

(2) 入力が終了したら，使用しているプログラム言語のコンパイラによって，ソースプログラムを機械語に翻訳させる．この段階ではソースプログラムは命令の指示書として扱われているのでなく，単に機械語に翻訳すべき文字や記号の列として扱われているに過ぎない．コンパイルが終了するとコンパイラの出力は2進数字の列としてのオブジェクトプログラムとなる．

(3) このオブジェクトプログラムを実行させるには，計算機に「実行」の指示を与える．

(4) 命令の実行に必要な通常の意味でのデータを，どこからか(たとえばキーボードから)与える．

以上の手順を概念図に示すと図1.3のようになる．こうした何段階かの仕事を1つの計算機システムでスムーズに実行できるのは，オペレーティングシステムのもとで整然と各種の処理が実行されていくからである．

(c) パーソナル・コンピュータ

近年，パーソナル・コンピュータの著しい機能向上に伴って，計算機の入門教育にパーソナル・コンピュータが利用されるケースが激増している．本書も，それを前提としているのでパーソナル・コンピュータに関する基本的事項を概説しておく．

われわれが見聞するコンピュータにはさまざまな呼称が付されている．計算機システムの規模と，

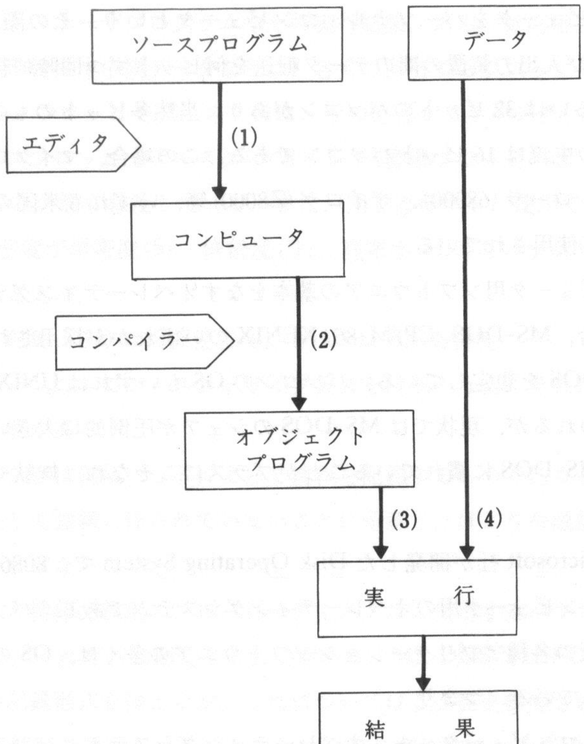


図 1.3　プログラム実行の手順

その利用目的(主たる機能)等を勘案して，マイクロ・コンピュータ，ファミリー・コンピュータ，パーソナル・コンピュータ，ミニ・コンピュータ，オフィス・コンピュータ，スーパーミニ・コンピュータ，汎用中型・大型コンピュータ，スーパー・コンピュータ，などと分類される．これらの間に，規模や性能上の明確な分類基準が定義されているわけではなく，マイコンとファミコン，あるいはミニコンとオフコン等の間に，しばしば規模や性能の逆転もありうる．

計算機の基本構成として，これまでに(a)および(b)項で述べてきた内容は，これらのコンピュータ群の大部分に共通している．ただ，使用目的に応じてCPUの処理速度，主記憶容量，補助記憶装置の規模と種類，情報通信の速度や能力，オペレーティングシステムの機能と規模，主として利用されるソフトウエア，などがそれぞれ大幅に異なっているわけである．

さて，IC(Integrated Circuit, 集積回路)やLSI(Large Scale Integrated Circuit, 大規模集積回路)技術の進歩に支えられて，1970年代から電子論理演算による演算機構を数ミリ四方のシリコンチップ上に組み込んだマイクロプロセッサが実用化され，各方面で利用されることとなった．現在実用されているマイクロプロセッサには4ビット，8ビット，12ビット，16ビット，32ビットなどのものがあり，小さいものはクーラーやエアコン，自動車の燃料制御装置，オーディオ機器等々に組み込まれて広く利用されている．

このマイクロプロセッサをCPUの一部として使用し，メモリーや入出力装置を備えた，ある程

度汎用性のあるコンピュータをパーソナル・コンピュータという．その際，CPUでの演算や，CPUと記憶装置および入出力装置の間のデータ転送を何ビットずつ同時に行なうかによって，8ビット，16ビットあるいは32ビットのパソコンがあり，当然多ビットのものほどハードウエア上の機能は高い．現在の主流は16ビットパソコンである．この場合，マイクロプロセッサとしてはインテル(8086)，モトローラ(68000)，ザイログ(Z8000)等，いずれも米国の企業で開発されたもののどれかが主として使用されている．

パーソナル・コンピュータ用ソフトウエアの基本をなすオペレーティングシステムとしては，16ビットパソコンの場合，MS-DOS, CP/M-86, XENIXのいずれかが採用されるケースが多い．本書では，以下，MS-DOSを想定している．パソコンのOSもいずれはUNIX系のものへ次第に移行していくものと思われるが，現状ではMS-DOSのシェアが圧倒的に大きい．将来，UNIX系に移行する場合でも，MS-DOSに慣れていることがプラスにこそなれ，無駄やマイナスになることはないであろう．

MS-DOSは米国Microsoft社が開発したDisk Operating Systemで，8086系のプロセッサを搭載したパーソナル・コンピュータ用のオペレーティングシステムである．パソコン上で利用できる言語プロセッサや市販の各種アプリケーションソフトウエアの多くは，OSの諸機能を利用することを前提として作られている．つまり

パーソナル・コンピュータ ＋ オペレーティングシステム

ではじめてユーザの各種ソフトウエアが利用可能になると考えればよい．

図1.4はソフトウエアの階層状構成を模式的に示したものである．われわれユーザは，ハードウエアに最も密着した基本ソフトたるオペレーティングシステムの上で，エディタや言語プロセッサを用いてユーザプログラムを作成し計算処理を行なったり，あるいはワードプロセッサや各種の作表・作図用等の応用ソフトウエアを用いて必要な作業を実行することができる．

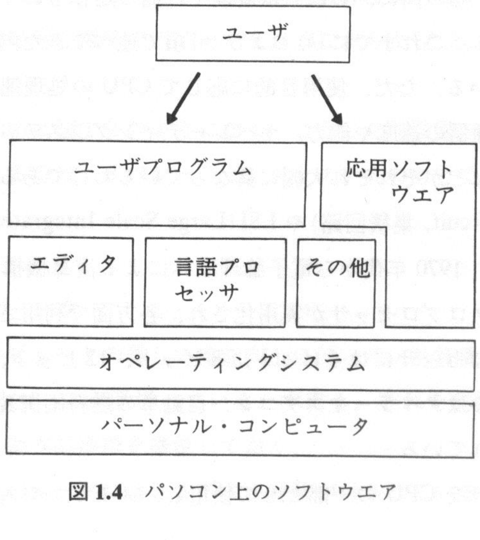


図1.4 パソコン上のソフトウエア

さて，図1.4で個々人のユーザプログラムやその実行結果，あるいは応用ソフトウエアを用いて行なった各種の情報処理結果や中間データなどは，各人の所有するフロッピーディスク上に記録保存しておくことになる．フロッピーディスクには，その直径が3.5インチ(マイクロフロッピーディスク)，5インチ(ミニフロッピーディスク)，8インチ(標準フロッピーディスク)のものがあり，現状では5インチのものが最も普及している．また，同じ直径でも，使用できる面が片面(1)か両面(2)か，および記録密度が単密度(S)，倍密度(D)，高密度(HD)のいずれであるかの違いにより多くの種類がある．たとえば,

1Sは片面単密度，2Dは両面倍密度，2HDは両面高密度

を意味する．使用するパソコンのディスクドライブ装置を確認のうえ，それに適合するフロッピーディスクを購入する必要がある．

またフロッピーディスクは磁気ヘッドによって情報を記録するものであること，比較的薄くて柔らかい円盤であって決して強靱に作られていないことに留意し，ほこりや磁気から遠ざけるなど，取り扱いに注意しなければならない．

なお，新たに購入した情報記録用のフロッピーディスクは，使用するパソコンのオペレーティングシステムに合わせて，それぞれ初期化する必要がある．MS-DOSではFORMATというコマンドを用いてディスクの記録形式を整えるが，これについては2.5節を参照されたい．

最後に，計算機入門教育で用いられるパソコンは，それぞれ独立した形で使用(スタンドアロン使用)される場合もあれば，パソコン間に通信回線を張っていわゆるLAN(Local Area Network)を組み，このLAN内にサーバやスーパバイザ(多くの場合にこれらもやはりパソコンまたはミニコン)を組み込んで，さまざまな授業支援機能を持たせることも広く行なわれている．さらに，このようなパソコン間のLANの他に，汎用の大型計算機を装備し，これにパソコンを接続して，パソコンと大型計算機とを仕事の内容に応じて使い分けるなどのことも行なわれている．この場合，大型計算機の方をホスト・コンピュータとかメインフレームなどと呼び，パソコンの方は端末(ターミナル)とかワークステーションと呼ぶ．ある程度の入出力機能と処理能力を持った端末をワークステーションと呼ぶとすれば，ホスト・コンピュータに接続されたパソコンは立派なワークステーションということになるわけである．

1.2　プログラミングとは

プログラミングを初めて学習するに際しては，学ぶべきことがらが少なくはない．学習すべき個々の内容そのものは，比較的単純なことであるが，単純なことであってもそれらが重なりあうと，結構複雑で難解であるかのような印象を受けるかもしれない．そこで，プログラミングとその実習に必要なことがらのうち，未知の部分をある程度意識的に整理・分類して，必要以上の混乱に陥らぬよう留意した方がよい．本節では計算機を用いた問題解決のプロセスに関する簡単なモデルを考え，それに沿って，プログラミングや計算機の利用が自由自在に行なえるまでに学習すべきことが

らを整理し，本書の守備範囲と内容の構成について解説する．

(a) 計算機を用いた問題解決

計算機によって何らかの問題解決をはかろうとするとき，そのプロセスをモデル化すると図1.5のように考えることができる．

このうちプロセス A_1 および A_2 に関しては，どのような分野の問題であれ，われわれ人間の最も高度な判断力と洞察力を必要とする部分であるということができる．分野によってはモデル化・記号化・定式化などが不可能な問題群も多い．当然，すべての問題がコンピュータによって解けるわけのものではなく，モデル化・記号化・定式化等がある程度行なえる場合に，はじめて計算機を用いて問題解決を試みることになる．

問題を発見する能力，問題点を的確に把握する能力，さらにはその問題の本質にかかわるエッセンシャルな部分をつかみとった良いモデルを構築する能力などは，創造性や独創性に負うところが大きく，広範な基礎学力の積み重ねの上に，ある程度の研鑽を経てはじめて身につくものであろう．これらの部分が本書の守備範囲の外にあることは言うまでもない．本書は図1.5のB以下のプロセスについて学ぶためのテキストである．したがって，本書に収められている例題はいずれも，すでにモデル化ないし定式化された段階にあるものを題材にして，B以降のプロセスを実行するため

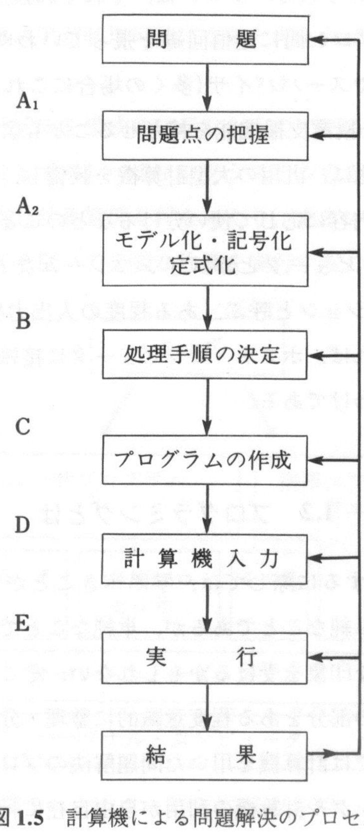


図 1.5 計算機による問題解決のプロセス

の演習課題として提示されている．なお，広い意味でのプログラミングとはB以降の全プロセスをさすと考えてよいであろう．

(b) アルゴリズム

プロセスBはモデル化・記号化・定式化が終わった後，それらを用いて広い意味での「計算」を行なおうとするときの計算方法や計算手順を決定するプロセスである．計算機によって問題を解くための具体的手順であってアルゴリズムと呼ばれる．簡単な問題なら，それを解くアルゴリズムも単純な命令の集まりで記述できる．問題が複雑で大規模なものである場合には，これをより単純で小さい問題群に分割し，その上で個々の小問題についての詳細手順を決定していくというトップダウン方式の段階的記述方法をとるのがよい．本書では，コンピュータによる「計算」にかかわる基本的なアルゴリズムを第3章や第Ⅱ部4章以降で例題を通じて紹介する．

(c) プログラミング

プロセスCは，プロセスBで決定したアルゴリズムに沿って，計算機に実行させるべき具体的計算・処理を，計算機に通用する言語で逐一記述するプロセスである．これが狭い意味のプログラミングにほかならない．このためには何らかのプログラミング言語を一つは知っている必要がある．

問題分野に応じて，それぞれに適したプログラミング言語が用いられる．科学技術計算の分野ではFORTRAN, PASCAL, Cなど，事務計算ではCOBOL, FORTRAN, BASICなど，人工知能や知識工学の分野ではLISP, PROLOG, PASCAL, C等々である．

本書ではPASCAL(TURBO Pascal)を採用した．その理由は，(1)構造化されたわかりやすいプログラムを書くのに適した言語であること，(2)豊かなデータ型やデータ構造の指定ができるためさまざまな情報処理の分野に適用可能であること，(3)パーソナル・コンピュータ上での処理を考えるとき，TURBO Pascalは極めて簡便で機能も十分に高く，入門教育用としても最適と考えられること，(4)「手続き」や「関数」の再帰的呼び出しが可能であること(第3.5節参照)，(5)より実用的なプログラミング言語CやAdaへの移行がスムーズに行なえること，などによる．

PASCALの文法やデータ構造に関しては，本書の1.3節でその初歩的な例を，第3章で基本文法を，第4章以降では実用的な文法やデータ構造の利用法を，それぞれいずれも例題を通じて述べる．

(d) 操　作　法

プロセスDではプログラムを計算機に入力する．これ以降については，すでに1.1節(b)項で述べたとおりである．

このプロセスでは計算機システムの利用手順やキーボードの操作法に関する知識が必要となる．これらは初心者が結構とまどう部分なのだが，従来個々の計算機システムによって，その操作法や利用に至る手順が大幅に異なっていたため，一般的な記述はきわめて困難であった．しかし，16

ビットパソコンの普及とともに，そのOSもある程度整理体系化されてきており，本書ではMS-DOS上のTURBO Pascalを想定して，第2章でTURBO Pascalの操作法について，その基本的な部分だけを述べてある．

なお，タイプライタを使う機会が少ないわれわれ日本人には，キーボードに慣れることもまた，計算機と付き合っていくうえで始めに払わなくてはならない努力の一つである．しかし，最初は厄介なようでも，しばらくすれば必ず慣れて苦にならなくなるものであるから心配はいらない．

〈キーボードについて〉

パソコンのキーボードは通常，図1.6に示すようなキー群から構成されている．

多くのキーボードでは，「普通文字キー」は白色，「制御コードキー」と「モードキー」は灰色などの着色キーとなっている．

フルキー： まずフルキーの部分について，そのキー配置の1例を図1.7に示す．「普通文字キー」の配置は通常のタイプライタと同様である．これらのキーによる入力に際しては，モードキーによって次の3つの入力モードのいずれかを選択する．以下では主として英文字入力モードのみを使用する．

英文字入力モード　白色キーの左半分(または左上半分)に示されている英文字や記号を入力するモード．

かな文字入力モード　白色キーの右半分(または右下半分)に示されているかな文字や記号を入力するモード．

グラフィック入力モード　キーには示されていない特殊な模様やパターンを入力するモード．日本語処理(2.5節(d)項参照)を利用するシステムではこのモードを使用できない場合もある．

パソコン起動時には，通常，英文字入力モードとなっているはずである．試みにフルキー中央付近の任意の白色キーを1つだけ押してみるとよい．そのキーに記されているアルファベットの小文字が画面に表示されれば，英文字入力モードとして正常である．大文字のアルファベットが表示されるようならCAP(またはCAPS)キーが押下された状態にあるので，再度CAP(またはCAPS)

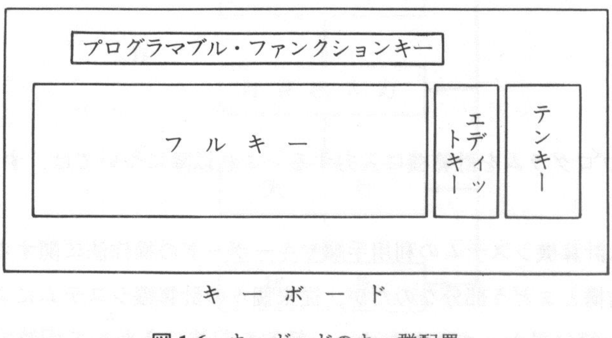


図1.6 キーボードのキー群配置

キーを押して通常の状態に戻す．また，もし白色キーの右半分に記されている，かな文字が画面に表示されるようなら，かな文字入力モードとなっているのであるから，かなキー(またはカナキーあるいは英数字キー)を押して英文字入力モードに直す．

英文字入力モードでは白色キーの働きは通常のタイプライタと変わらない．すなわち，各キーを押せば，そのキーに示されている英字の小文字や数字，あるいはそのキーの左下部分に示されている記号が入力される．またSHIFTキーを押しながら白色キーを押せば英文字のシフト入力モードとなり，各キーに示されているアルファベットの大文字，あるいはそのキーの左上部に示されている記号が入力される．

英文字入力モードでCAPキー(またはCAPSキー)を押すと，それ以後の英字入力は全て大文字となる．このキーは英字以外のキーに対しては作用しない．

フルキーの中の制御コードキーでは，次のようなキーをしばしば使用する．(　)内は本書で用いる略記号を示す．

リターンキー(RET)：　フルキーの右端中央部分にある大きなキーで曲がった矢印が表示されている．通常，キーボードから入力された文字列やデータはRETキーが押されて初めて，その行の内容がシステムに渡される．

バックスペースキー(⇦)：　RETキーの上にある．カーソルの左の1文字を削除する．

スペースキー：　空白(ブランク)を入力するための細長いキー．

シフトキー(SHIFT)：　フルキーの左右両端にある．このキーを押下しながら他のキーと組み合わせて用いる．

コントロールキー(CTRL)：　左側のシフトキーの上にある．このキーも他のキーと組み合わせて制御コードを発生させるのに用いる．

エスケープキー(ESC)：　フルキー左上端にある．

フルキーの操作に際しては次の点に留意するとよい．

(1)　両手の指は常にホームポジションに置くよう心掛ける．

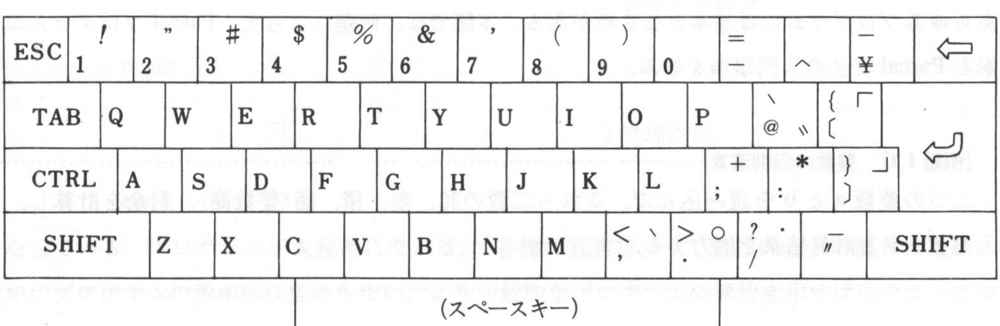


図1.7　JISによるフルキーの配置例

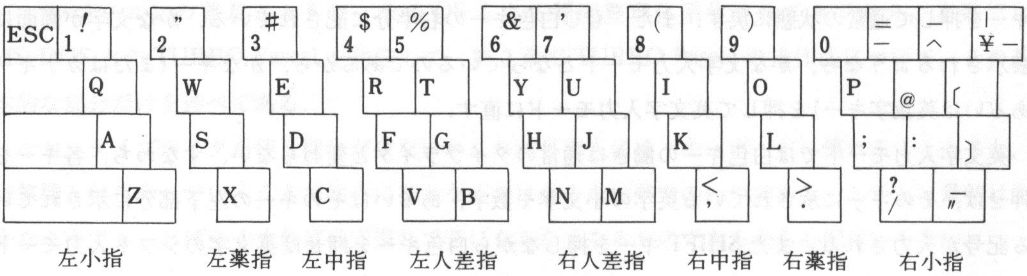


図 1.8 各指のキー分担

ホームポジションでは，左手の小指から人差指までをそれぞれA，S，D，Fキーの上に置き，右手は人差指から小指までをそれぞれJ，K，L，;キーの上に置く．

(2) 両手各指のキー分担を決めて，全部の指を使うよう心掛ける．

キーボード上のキーを図1.8のように，左上から右下に至る列ごとにグループ化し，各指の分担とする．親指はスペースキーやカナ漢字変換キーなどを分担する．

その他のキー群には次のようなものがある．

プログラマブル・ファンクションキー： フルキーの上方に並んでいるキーで，よく使う命令などを記憶させ，1ストロークで命令を起動させるなどの目的に使う．

エディットキー： フルキーの右方に，2列5段程度のキーがあり，プログラムや文書の編集作業に利用する．

テンキー： エディットキーより更に右方に，0から9までの数字や四則演算記号などのキーがあり，10進の数値を入力するときなどに使う．これらのキーは，フルキーの中にあるものと同じ意味を持っている．

1.3 Pascal 入門

(a) プログラムの基本形

あらゆるプログラムには基本となる形がある．本節では，例題によって，Pascalプログラムの基本形とPascal文法の入門事項を学ぶ．

[例題 1.1] 整数の四則演算

二つの整数aとbを読み込んで，これら二数の和，差，積，商(整数商)，剰余を計算し，入力データと計算結果を出力する．

この場合，プログラムが実行すべき処理内容は

(1)　データaとbの読み込み(データ入力)

(2)　和，差，積，商，剰余の計算(必要な計算処理)

(3)　入力データおよび計算結果の出力(データ出力)

の三つに区分けできる．

プログラミング言語は，これらのデータ処理手順を書く際の約束ごとの集まりである．Pascalの約束ごと(Pascal文法)に則って，上の処理手順をプログラム化した例をプログラム1.1に示す．

プログラム1.1

```
program seisuu_sisoku(input,output);   { プログラム頭部 }
  var a, b : integer;                   { 変数の型宣言 }
      wa, sa, seki, shou, amari : integer;    { 同上 }
begin                                   { 実行部分の始まり }
  readln(a,b);                          { データの読み込み }
  wa    := a + b;
  sa    := a - b;
  seki  := a * b;                       { 二数の乗算 }
  shou  := a div b;                     { 整数の除算 }
  amari := a mod b;                     { 整数除算の剰余 }
  writeln(a:7,b:7);                     { データの書き出し }
  writeln(wa:8,sa:8,seki:8,shou:8,amari:8)    { 同上 }
end.                                    { プログラムの終り }
```

プログラム1.1の骨格は

```
program　プログラム名(input, output);
    var　変数名, ……, 変数名 : 型名;
          ⋮
          ⋮                  } 各種の宣言等
          ⋮
    begin
          ⋮
          ⋮                  } 処理内容
          ⋮
    end.
```

となっている．プログラムが大規模になれば，「各種の宣言等」や「処理内容」の部分は複雑になるが，プログラムの基本形は変わらない．より一般的なプログラムの構造を示せば次のようになる(各宣言や定義の具体的な方法，意味機能等については第3章以降で述べる)．なお，各種宣言・定義部分と実行部分はまとめてブロックと呼ばれる．

```
program  プログラムの名前(ファイル名, ・・・, ファイル名);

    label ································ラベル宣言部
    const ································定数定義部
    type ·································型定義部          各種宣言・
    var ··································変数宣言部        定義部分
    procedure ····························手続き宣言部
    function ·····························関数宣言部

    begin
                                          実行文部          実行部分
    end.
```

ここに英字で記した program, label, const 等々の語は，Pascal における予約語(本節末尾参照)であって，これらの用語を用いる場合には，定められたとおりの綴り記号を使用しなければならない．また，プログラム 1.1 における input, output, integer, readln, writeln などは，特定の意味や機能を持たせてあらかじめ言語処理系に用意されている標準の名前である．これらの標準名としては，標準型名(integer, real 等)，標準定数名(maxint, true, false 等)，標準変数名(input, output 等)，標準手続き名(readln, writeln 等)，標準関数名(abs, exp, sin, cos 等)が定義されており，それぞれの名前によって与えられる機能を利用することができる．これらについても今後必要に応じてより詳しい説明を加えていく．

以下では，Pascal 文法への入門を兼ねて，プログラム 1.1 の各行が何を意味しているかを説明しよう．なお，Pascal プログラムで { と } で囲んだ中味は，注釈ないしはコメントであって，プログラムの実行内容には何等影響を与えない．プログラム 1.1 でも，これを利用して，簡単な説明を試みてある．

(b) プログラム 1.1 の説明

(1)行目： Pascal プログラムの始まりであり，プログラム頭部と呼ぶ．「seisuu_sisoku」はこのプログラムの名前である．小かっこ内の「input」および「output」はそれぞれこのプログラムでデータの「入力」および「出力」が行なわれることを意味している．文の区切りには区切り記号 ; (セミコロン)をつけなければならない．

(2)行目：変数の型宣言部.

計算機におけるデータ処理においては，入力したデータはすべていったん記憶域にしまわれる．記憶域内のデータは必要に応じてCPUに取り出されて，処理が施される．あとで必要となる処理結果等もいったん記憶域にしまっておく．これら記憶域の中の記憶場所はプログラムの中で，逐一番地で指定する代わりに，データに名前をつけることによって確保する．確保した記憶域の中味そのものは，入力するデータや計算処理により変わるので「変数」と呼び，その名前を「変数名」という．

Pascalプログラムではプログラムの中で使用するすべての変数名を前もって宣言するとともに，おのおのの変数が表わすデータの型を明示する．

1.1節(a)項で述べたように，計算機の内部においては，すべてのデータや，記号，文字，命令などは，2進数字の列で表現されていると考えることができる．これら2進数字の列が表わす実質的内容は，その2進数字の列が，どのようなタイプのデータを表わしているかを規定しないと定まらない．Pascalでは，さまざまな型のデータを自ら定義し使用することができるが，当面は標準的に用意されている次の4つの基本的なデータの型を挙げておく．

integer (整数型)：-32768～32767の範囲の整数*1
real (実数型)：2.9×10^{-39} から 1.7×10^{38} までの10進数で，有効数字約11桁程度の数*1
char (文字型)：ASCII形式の文字*2
Boolean (論理型)：true, false の論理値を持つ

*1：TURBO Pascalでは整数は2バイト，実数は6バイトを占める．一般のPASCALでは処理系によって相違するので注意を要する．

*2：ASCII(American Standard Code for Information Interchange)

プログラム1.1 (2)行目の

```
var a, b : integer ;
```

は，変数aおよびbの記憶域を確保するとともに，その記憶域にしまわれるデータが整数型であることを宣言している．このように，変数宣言では，最初に予約語varを置き，1文字分以上の空白をおいて，カンマで区切られた同じ型の変数名の並びが置かれ，その後にコロン，型，セミコロンが続く．

(3)行目：(2)行目の続きで，計算結果を記憶しておく5つの変数記憶域を確保している．

(2)行目と(3)行目はまとめて

```
var a,b,wa,sa,seki,shou,amari : integer ;
```

としてもよい．

なお,「変数名」や「プログラム名」など，各種の「名前」は次の規則を満たすものであれば，プログラム作成者が自由につけることができる．ただし，予約語と同じ名前を使用することはできない．

・英字で始まり，英字または数字をならべたもの．

・127文字以内(処理系によっては最初の6文字～数十文字のみを認識)

・TURBO Pascalでは特殊記号の下線「_」も使用することができる．

多くのPascal処理系では，英字の大文字と小文字は，文字列データの中を除いて区別しないので，どちらを用いてもよいし，両者を適宜混用してもよい．

(4)行目：「begin」はプログラム実行部分の始まりを意味する．この「begin」と(13)行目の「end」にはさまれた部分がこのプログラムの処理実行部分である．ここには実際に行なう動作を記述するが，動作記述の単位となるものを文と呼ぶ．文には他の文を含まない単純文と，内部構造を持つ構造文とがある．文と文の間はセミコロンで区切る．

(5)行目：「readln(a,b)」は，計算に先立って，どこからか与えられる二つの数値をそれぞれ順に変数aとbに代入することを意味している．このようなデータの読み込みは，プログラム実行時にさまざまな媒介(入力装置)を経由して行なうことができるが，その方法は後述する．データの読み込みにはread(　)とreadln(　)を標準的に用いることができる．(　)内にはデータを与えたい変数名を，コンマで区切って必要なだけ並べることができる．read(　)では，(　)内に指定した変数の個数だけデータが読み込まれ，順次各変数に代入される．

(6)行目：変数aとbに入っている二つの数値を足しあわせ，結果を変数waにしまう．結果を変数にしまうことを「代入」するといい，Pascalでは代入を意味する記述として「:=」(コロンイーコール)を使う．この「:=」を使う文を代入文と呼ぶ．代入文では，右辺の計算結果を左辺の変数に代入する．

(7)行目～(10)行目：整数の四則演算とその結果の「:=」による代入である．和と差の演算を表わす記号は通常の数学記号と変わらないが，積の演算記号としては(8)行目「*」(アスタリスク)を，また整数の除算には「div」を，剰余を求めるには「mod」を使う．

7　div　3　の演算結果は2

7　mod　3　の演算結果は1

(11)行目：(5)行目で読み込んだ入力データをそのまま出力している．意図通りに正しくデータが入っているかどうかの確認の意味からも，計算結果だけでなく，入力データも改めて出力しておくのがよい．

データの出力もさまざまな媒介(出力装置)を通じて行なえるが，当面はディスプレイ上への出力を想定しておく．

「writeln(a:7, b:7)」は，変数aとbにしまわれている数値を，それぞれ7桁分の幅の中に右に寄せて書き出すことを意味する．書き出しにはwrite(　)または

writeln(　)を標準的に用いることができるが，write(　)では書き出したあと改行をしないのに対して，writeln(　)では(　)内のデータ書き出し終了後，改行を行なう．

入出力用のread, readln, write, writelnなどは前もって標準的に用意された手続き(3.6節参照)である．すでに定義されているユーザ定義の手続きや，あらかじめシステム側で用意されている標準手続きを呼び出す文のことを手続き文という．手続き文は手続き名そのものであり，手続き名の後ろに必要に応じて括弧に囲まれた引数(3.6節参照)を並べることができる．

(12)行目：演算結果の出力である．行の終わりに，命令，文，宣言等の区切り記号であるセミコロンがないのは，次の(13)行目に「end」があり，これも命令を区切る働きを持っているからである．ただし，TURBO Pascalでは，この場合にもセミコロンがあってもよい．

(13)行目：(4)行目の「begin」と呼応する形で「end」は実行部分の終わりを示している．更にプログラム全体の終りには．(ピリオド)を付さねばならない．

(c) Pascal で使用できる文字

Pascal プログラムで使用できる文字は次のように定められている.

英字

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

数字

0 1 2 3 4 5 6 7 8 9

特殊文字

+ − * / = < > ( ) { } [ ] . , : ; '

(ISO Pascal では上記特殊文字の他に | の使用も許しているが, TURBO Pascal では | の代わりに下記の ^ を使う.)

TURBO Pascal 固有の特殊文字

^ # $ _(アンダースコア) @(アットマーク)

これらの特殊文字は, 単独または組み合わせによって次のような意味を持つ.

算術演算

*	乗算(整数型および実数型)
/	除算(実数型)
+	加算
−	減算
( )	演算順序の制御

プログラム構成

:=	代入
.	プログラムの終わりを示す区切子
;	文の終わりを示す区切子
{ }	注釈やコメント またはコンパイラーへの指示(TURBO Pascalのみ, 第5章参照)
'	文字列の始めと終わり

比較・テスト

＝	等しい
＜	小なり
＞	大なり
＜＝	小または等しい
＞＝	大または等しい
＜＞	等しくない

また，英字を組み合わせたいくつかの単語は，1まとまりで1つの記号として用いられ，綴り記号とか予約語と呼ばれる．Pascal における予約語の役割は重要であって，以下に示す予約語は，ユーザが勝手に名前として再定義し，使用することは許されない．

予約語

*absolute　and　array　begin　case　const　div　do　downto
else　end　*external　file　for　forward　function　goto
if　*implementation　in　*inline　*interface
*interrupt　label　mod　nil　not　of　or　packed
procedure　program　record　repeat　set　*shl　*shr
*string　then　to　type　*unit　until　*uses　var
while　with　*xor

(あたまに*印を付けたものは TURBO Pascal 特有の予約言語である.)

(c) Pascal で使用する文字

Pascal プログラムで使用できる文字は次のように定められている。

英字

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

第2章 TURBO Pascalシステムの使い方

TURBO Pascal は，操作性に優れており，使い勝手の良いパーソナル・コンピュータ用 Pascal として定評がある．プログラムの作成や修正，プログラムのコンパイルと実行，更にはディスケットへのセーブ，ディレクトリの表示等が，「メニュー」に従って手軽に行なえるシステムなのである．

この章では，TURBO Pascal システムの起動，プログラムの作成や修正などの編集作業，コンパイル，実行，プログラムのディスケットへのセーブなど，一連の作業に必要な事項について述べる．なお，本書では MS-DOS 上の TURBO Pascal を想定して記述する．

2.1 TURBO Pascal の起動*

オペレーティングシステムのプロンプト(入力促進記号)

F：￥＞

は，ドライブ F をカレントドライブとして MS-DOS のコマンド入力待ち状態であることを示している．

TURBO Pascal を起動して，プログラムを自分のフロッピーディスクに保存したいときは，前もって

F：￥＞<u>a：⏎</u>

として，カレントドライブを A に変更する．ここに下線部はユーザが入力するものであり，⏎キーはリターンキーを押すことを意味する．するとディスプレイ上には

A：￥＞

とプロンプトが表示され，再びコマンド待ち状態となる．そこで，

A：￥＞<u>pascal⏎</u>

と入力すると図 2.1 のような画面になる．ファイル名(2.2 節参照)が決っているときは，pascal の後にスペースをあけてファイル名を指定する．例えば，ファイル名が SISOKU. PAS であれば，

A：￥＞<u>pascal sisoku⏎</u>

と入力する．

---

* ここでは東京大学教育用計算機センターにおける起動法について説明する．一般のパーソナル・コンピュータでは，たとえばドライブ A に MS-DOS のシステム，ドライブ B に TURBO Pascal のフロッピーディスクを入れ，B＞turbo⏎と起動する．

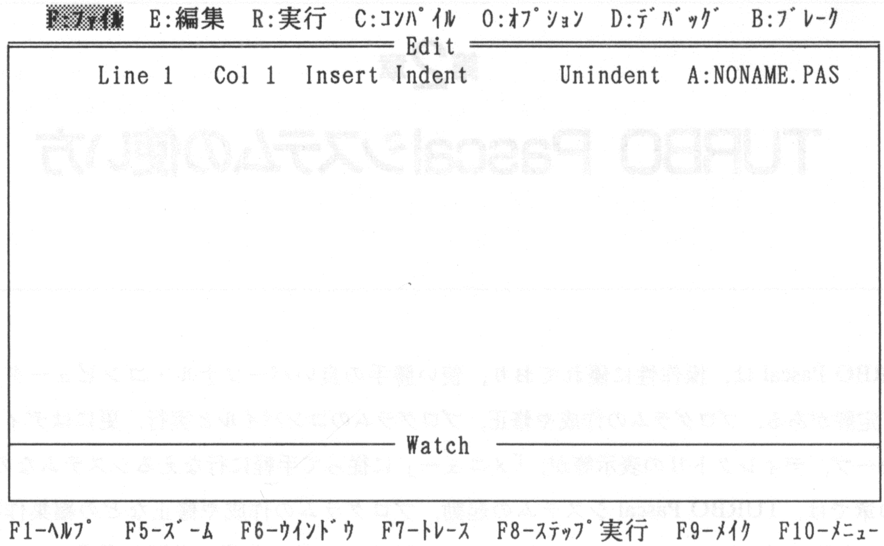


図 2.1 TURBO Pascal の初期画面

ファイル名を指定した場合，そのファイルが存在すれば画面にファイルの内容が表示され，ファイルが存在しなければ，プログラム作成状態となる．ファイルを指定しないときは，画面左上のF：ファイル が反転していることがわかる．

画面一番上の１行はメインメニューといって，主な操作の名前が並んでいる．

F：	ファイル	ファイルから読み込む，ファイルにセーブする，終了など
E：	編集	プログラムの作成・編集(エディタ)
R：	実行	プログラムの実行，実行画面の表示
C：	コンパイル	プログラムのコンパイル
O：	オプション	コンパイルの指定，環境設定
D：	デバッグ	プログラムのデバッグ用
B：	ブレーク	ブレークポイントの設定

これらの機能を選択するには，先頭のF, E, R, C…の文字をキーで押すか，←，→キーでカーソル(文字の反転)を移動し，↵キーで選択することができる．ただし，よく用いる操作はホットキーといってどの状態からもキー操作することができる．

2.2 ソースプログラムファイルの作成

ここでは，第１章で学んだ例題・プログラム 1.1 を使って，プログラムの入力方法および編集方法について述べる．コンピュータの中では一まとまりのプログラムやデータはすべてファイルとして保存される(それは丁度われわれが一まとまりの書類や資料を別々のファイルに整理してしまっ

```
F:ﾌｧｲﾙ  E:編集  R:実行  C:ｺﾝﾊﾟｲﾙ  O:ｵﾌﾟｼｮﾝ  D:ﾃﾞﾊﾞｯｸﾞ  B:ﾌﾞﾚｰｸ
═══════════════════════ Edit ═══════════════════════
    Line 1   Col 1  Insert Indent    Unindent  A:NONAME.PAS
program seisuu sisoku(input,output);⏎
  var a,b : integer;⏎
      wa,sa,seki,shou,amari:integer;⏎

─────────────────────── Watch ───────────────────────

F1-ﾍﾙﾌﾟ  F5-ｽﾞｰﾑ  F6-ｳｲﾝﾄﾞｳ  F7-ﾄﾚｰｽ  F8-ｽﾃｯﾌﾟ実行  F9-ﾒｲｸ  F10-ﾒﾆｭｰ
```

図 2.2　プログラムの入力

ておくのと同様である）．

Pascal のような高級言語で書かれたプログラムはソースプログラムまたは原始プログラムと呼ばれる．したがってプログラム 1.1 をコンピュータに入力することによって，1 つのソースプログラムファイルを作ることになるわけである．

まず，プログラムを入力・編集するためには，メニューの E：編集（エディタ）に入る．そのために，キーボードの E のキーを押すか，ESC キーを押すか，PF 10 キーを押す．どのキーを選ぶかは自分のやりやすいもの，あるいは覚えやすいものを選ぶ．編集状態に入ったかどうかはカーソルを見ていれば確認できる．

このキー操作を行うと，枠内の左上付近に小さい四角が点滅してテキスト入力状態に入ったことがわかる．プログラムにしたがって文字を打って，各行の終わりに⏎キーを打つと改行が行われる．なお，プログラム 1.1 の各行の後ろにある｛｝で囲まれた部分はコメントといってプログラムに直接関係のない部分なので必ずしも入力する必要はない．漢字の入力に慣れてきたら，適度にコメントを追加するとよい．

文字を入力していくと枠内の一番上の行の数字が変化することがわかる．この行をステータス行といい，次のような意味がある．

Line m　　プログラムの先頭から数えてカーソルが第 m 行目にあることを示す．

Col n　　画面の左端から数えてカーソルが第 n 列目にあることを示す．

Insert　　エディタによる文字入力には Insert と Overwrite がある．Insert の表示は Insert 状態にあることを示し，Overwrite のときは何も表示されない．Insert ではカーソルのある位置に入力文字が挿入され，カーソルの右にあったテキストは 1 文字ずつ右にずれていく．

Overwrite 状態ではカーソルの位置にある文字はキーボードから入力された文字に置き換えられる．モードの切り替えは後述の CTRL-V を使う．

Indent　自動段付けが行われることを示している．Pascal プログラムではプログラムを読みやすくするため段付け(Indentation)を多用する．したがって段付け機能は有効である．

A：filename

エディット中のファイルのドライブ名，ファイル名が表示される．

NONAME. PAS はまだ名前が指定されていないことを意味する．

2.3 プログラムの編集

キーボード操作に慣れないうちは入力ミスも多く，打ち込んだ文字や記号を修正する必要が度々生ずるものである．プログラムの修正や書き直しなども，同じエディタで行える．

エディタには，カーソルを自由自在に動かす，文字を挿入する，削除する，探索する，置き換える，等々の 46 個のエディットコマンドが用意されている．これらのコマンドは表 2.1 に示すように，

・カーソル移動に関するコマンド

・挿入・削除に関するコマンド

・ブロックに関するコマンド

・その他のコマンド

の 4 つのグループに分類することができる．これらのコマンドのすべてが直ちに使いこなせる必要はない．ここではプログラムの編集作業に必要な最小限のコマンド(表中の＊印付き)だけを紹介する．表で 2 文字のコマンド，例えば CTRL-K　CTRL-B とあるのは，CTRL キーを押したまま順に K つづいて B と押せばよい．

(a) 重要なコマンド

スクリーンエディタでは画面内の必要な場所にカーソルを移動させ，文字や文字列，行などの挿入・削除・書換え等の編集作業を行う．エディタのデフォルト・キー・ストロークにおけるカーソル移動コマンドは系統的に構成されており，キーボード左端付近に隣接して並んでいる次の 4 つの文字が最も基本となっている．

[E]

[S]　[D]

[X]

これら 4 文字のキーは CTRL キーの近くに位置しており，左手の小指で CTRL キーを押しながら，中指や人差指で簡単に入力できる．また，キーの位置関係から CTRL-E がカーソルを上に，CTRL-X は下に，CTRL-S は左に，CTRL-D は右に動かすということもわかりやすい．CTRL キーとこれら 4 つの文字の内の 1 つを押し続ければ，カーソルは画面の中を素早く動きまわる．

表 2.1　エディット・コマンド一覧表

・カーソル移動のコマンド		・挿入や削除のコマンド	
＊1文字左へ	CTRL-S	＊挿入／上書きモードの切替	CTRL-V
＊1文字右へ	CTRL-D	＊カーソル位置の1文字削除	CTRL-Gまたは削除キー
1語左へ	CTRL-A	＊カーソルの左の1文字削除	(⇦キー)
1語右へ	CTRL-F	カーソルの右の1語削除	CTRL-T
＊1行上へ	CTRL-E	＊1行挿入	CTRL-N
＊1行下へ	CTRL-X	＊1行削除	CTRL-Y
スクロール・アップ	CTRL-W	行末までの削除	CTRL-Q　CTRL-Y
スクロール・ダウン	CTRL-Z		
1ページ上へ	CTRL-R		
1ページ下へ	CTRL-C		
行の左端へ	CTRL-Q　CTRL-S		
行の右端へ	CTRL-Q　CTRL-D		
画面の先頭へ	CTRL-Q　CTRL-E		
画面の最後へ	CTRL-Q　CTRL-X		
ファイルの先頭へ	CTRL-Q　CTRL-R		
ファイルの最後へ	CTRL-Q　CTRL-C		
ブロックの先頭へ	CTRL-Q　CTRL-B		
ブロックの最後へ	CTRL-Q　CTRL-K		
前のカーソル位置へ	CTRL-Q　CTRL-P		
・ブロックに関するコマンド		・その他のコマンド	
ブロックの始点マーク	CTRL-K　CTRL-B	＊編集終了	CTRL-K　CTRL-D
ブロックの終点マーク	CTRL-K　CTRL-K	タブ	CTRL-I
1語マーク	CTRL-K　CTRL-T	自動タブのオン／オフ	CTRL-Q　CTRL-I
ブロック表示／非表示	CTRL-K　CTRL-H	行を最初の状態へ戻す	CTRL-Q　CTRL-L
ブロックの複写	CTRL-K　CTRL-C	文字列探索	CTRL-Q　CTRL-F
ブロックの移動	CTRL-K　CTRL-V	探索した文字列置換	CTRL-Q　CTRL-A
ブロックの削除	CTRL-K　CTRL-Y	前と同じ文字列を探索	CTRL-L
ブロックの読み込み	CTRL-K　CTRL-R	コントロール文字入力	CTRL-P
ブロックの書き出し	CTRL-K　CTRL-W		
ブロックの印刷	CTRL-K　CTRL-P		

・カーソルを1文字左へ　CTRL-S　または←キー
・カーソルを1文字右へ　CTRL-D　または→キー
・カーソルを1行上へ　CTRL-E　または↑キー
・カーソルを1行下へ　CTRL-X　または↓キー

カーソルの移動以外に重要なコマンドに次のようなものがある．

・挿入／上書きモードの切り替え　CTRL-V
・カーソルの左の1文字削除　⇦キー(バックスペースキー)
・カーソル位置の1文字削除　CTRL-G　または削除キー
・カーソル位置に1行挿入　CTRL-N
・カーソル位置の1行削除　CTRL-Y

編集を終了してメニュー行に移るには，CTRL-K，CTRL-D と押すか，PF10 キーを押せばよい．ただし，TURBO Pascal ver. 5 では主要な操作が編集中に行えるので，必ずしもメニュー行に移る必要はない．

(b) ヘルプ

キー操作を忘れたときは，ヘルプを見て参考にすることができる．

ヘルプを見るには PF1 キーを押すが，例えば，プログラム作成・編集中はエディタのヘルプが(図 2.3)，エラーメッセージが出ているときはそのヘルプが出てくる．

```
F:ﾌｧｲﾙ  E:編集  R:実行  C:ｺﾝﾊﾟｲﾙ  O:ｵﾌﾟｼｮﾝ  D:ﾃﾞﾊﾞｯｸﾞ  B:ﾌﾞﾚｰｸ
═══════════════════════════ Edit ═══════════════════════════
  Line 1   Col 1  Insert Indent       Unindent  A:EX1.PAS
program ex1;
begin                ────────── Help ──────────
  writeln('PAS     基本カーソル移動コマンド
end.              ──────────────────────────────────
                  1文字左へ          CTRL-S または ←
                  1文字右へ          CTRL-D または →
                  1単語左へ          CTRL-A
                  1単語右へ          CTRL-F
                  1行上へ            CTRL-E または ↑
                  1行下へ            CTRL-X または ↓
                  スクロールアップ   CTRL-W
                  スクロールダウン   CTRL-Z
───────────────────────────────────────────── 次行 ──

F1-目次   ↑↓-ｶｰｿﾙ移動  Return-選択  ESC-ﾍﾙﾌﾟ終了
```

図 **2.3** 編集中のヘルプ画面

ヘルプ画面は中央の小さい枠で囲まれた部分であるが，右下に'次行'と出ることがある．これは，まだヘルプ画面が続くということを表している．↵キーの少し右上に'次行'と書かれたキーがあるので，そこを押すと次の説明を見ることができる．次の説明から前の説明に戻るには'前行'キーを押す．

ヘルプを見るのを中止したいときは，ESC キーを押す．また他の項目のヘルプを見たいとき，たとえば編集中に編集以外のヘルプを見たい場合は，PF1 を 2 回押して，図 2.4 のようなヘルプインデックスを出す．

次に←→↑↓のキーでカーソル(反転部分)を移動して，↵キーを押し必要な項目を選択する．

例えば，if, for, while などの文についての構文を知りたいときは，'Reserved word' を参照する．また read, write について詳しく知りたいときは，'Library procedures' を参照する．実行やセーブのキー操作を忘れたときは，'Hot keys' などを参照すれば必要な情報が得られる．

```
  F:ﾌｧｲﾙ  E:編集  R:実行  C:ｺﾝﾊﾟｲﾙ  O:ｵﾌﾟｼｮﾝ  D:ﾃﾞﾊﾞｯｸﾞ  B:ﾌﾞﾚｰｸ
═══════════════════════ Edit ═══════════════════════
    Line 1   Col 1  Insert Indent        Unindent  A:EX1.PAS
program ex1;
begin              ──────────── Help ─────────────
  writeln('PAS                I N D E X
end.               Help on Help           Hot Keys
                   Editor                 Error Messages
                    block commands        Installation(TINST)
                    cursor movement       Library functions
                    insert & delete       Library procedures
                    miscellaneous         Menu commands
                   Command-line options   Reserved Words
                   Compiler directives    Units
                   Debugging              Windows

F1-目次  ↑↓-ｶｰｿﾙ移動  Return-選択  ESC-ﾍﾙﾌﾟ終了
```

図 2.4　ヘルプインデックス

2.4　プログラムの翻訳・実行・終了

(a)　プログラムの翻訳・実行

エディタを終えてコンパイル・実行するには，CTRL-PF9 を押す．CTRL キーを押したまま，PF9 を押す．

プログラムに誤りがあると，枠内の一番上に赤い字で，コンパイルエラーが出てくる．エラーメッセージの意味がわからないときは，PF1 を押してヘルプを呼んで説明を見ることができる．

エラーの原因がわかればエディタで修正し，再度 CTRL-PF9 を押す．コンパイル・エラーが出ないと実行が始まる．プログラム 1.1 の場合，データとして 2 つの整数を 1 行で与えることになっている．そこでキーボードから，たとえば，

17　　6↵

のように 2 つのデータをスペース(ブランク)で区切って打ち込んでリターンキーを押すことによりデータが確定し，計算が実行される．ただし一瞬のうちに画面が元に戻る．このプログラムの実行結果は，もう一つの画面(MS-DOS)に表示されるので，それをあらためて見てみる．それには，ALT-PF5 を押す．ALT キー(または GRPH キー)は，スペースキーの少し左の方にある．これを押したまま，PF5 を押す．すると実行結果の画面が出てきて四則演算の結果を見ることができる．ここで何かのキー(たとえば↵キー)を押すと，エディタの画面に戻る．

(b) プログラムの保存・終了

プログラムは保存しないとすべて消えてしまい，何も残らない．プログラムの保存はPF2である．これを押すと，次のような表示が出てくる．今までファイルの名前を何も指定しなかったので，仮の名前 'NONAME' が付いており，正しい名前を聞いてくる．

```
──────── Rename NONAME ────────
A:¥NONAME.PAS
```

programの名前がseisuu_sisokuなので，ファイルの名前をSISOKUにする．sと打つと，枠の中の名前がすべて消え，sの文字が入る．続けてisokuと打つと，sisokuとなるので，⏎キーを押す．すると，フロッピー挿入口の赤いランプが2，3秒光って書き込みが行われたことがわかる．またEdit画面最上行の右端のファイル名がA：SISOKU. PASに変わっていることがわかる．

プログラムを作成しているときは，時々PF2を押してプログラムを保存する方がよい．思わぬ障害のために，長い時間をかけて打ち込んだプログラムが消滅してしまうこともあるからである．

TURBO Pascalを終了するときは，ALT-xを押す．初めにALTキーを押したままにする．すると，画面の一番下の表示が変わり，それを見るとxを押すと終了することがわかる．ALTキーを押したままxを押すと，終了してMS-DOSに戻る．

もしプログラムをエディタで修正して，それを保存していない場合，次のようなメッセージが画面に出てくる．

```
════════════ Verify ════════════
A:¥SISOKU.PAS not saved.  Save? (Y/N)
```

普通はyを押すが，必要のないときはnを押す．nを押すとセーブされないまま終了する．したがって終了しないで作業を続けるときは，ESCキーを押す．

(c) 計算結果の印刷

プログラム作成し，正常に実行できたことを確かめ，次の操作を行う．操作はすべてMS-DOSで行うので，Turbo Pascalは終了する．したがってALT-xを押して終了する．MS-DOSのプロンプトA：¥＞が出てくるので，実行型ファイルを作成する．

A：¥＞tpc　sisoku⏎

この操作により，SISOKU. PASというソースプログラムから実行型ファイルSISOKU. EXEを作成する．次にこれをMS-DOSで実行する．

A：¥＞sisoku⏎

前に実行したように2つの数値を入力して⏎キーを押すと計算結果が画面に表示される．次にもう一度実行し，実行結果をプリンタに出力する．

A：¥＞sisoku＞prn⏎

同じように2つの数値を入力する．ページプリンタの場合，印刷を行っても紙が排出されないことがある．そのときは，

A：¥＞ff↵

と入力すればよい．

2.5　MS-DOSの基本的コマンド

MS-DOSは磁気ディスクを備えたワークステーションやパーソナル・コンピュータの利用を援助するオペレーティングシステムである．TURBO PascalはMS-DOSの諸機能の一部を利用して，これまで述べてきたような一連の作業を巧みに処理しやすくしたシステムである．本節では，当面必要になる最小限の範囲でMS-DOSのコマンド等について説明する．詳しくはMS-DOSの専門解説書を参照されたい．

(a)　フロッピーディスクの初期化：FORMAT

新しいフロッピーディスクをMS-DOS上で使用するためには，最初にそのディスクを初期化して記録形式をMS-DOS用に整える必要がある．

(1)　フロッピーディスクを購入したら，まずユーザ用ラベルに，自分自身でそれと識別できるような標題等を記入して，ディスクの表側・メーカーラベルの横に貼付しておく．

(2)　フロッピーディスクは，ラベル貼付部分を手前かつ上側（縦長のスロットでは左側）にして，ディスクドライブのスロットに挿入する．カチッという音（または手応え）がするまで挿入したら，レバー等を閉じる．

(3)　MS-DOSのコマンド待ち状態では，カレントドライブ名と不等号「＞」からなるプロンプトが表示されている．この状態にあれば，またこの状態にあるときに限り，MS-DOSのコマンドが受け付けられる．なお，MS-DOSのコマンド入力においては大文字と小文字の区別はない．

(4)　ディスクの初期化にはFORMATコマンドを用いる．いま挿入した新しいディスクはドライブBにあるものとして，まずカレントドライブがF（MS-DOSのシステムファイルが入っているディスクのドライブ）であることを確認する．そうでないときは　f：↵　とキーインしてカレントドライブをFに変更する．

(5)　MS-DOSのプロンプト　F：¥＞　に対し

F：¥＞format　b：↵　　（イ）

（MS-DOSシステムファイルの中のFORMATというプログラムを用いてドライブBのディスクを初期化せよ）

または

F：¥＞format　b：/s↵　　（ロ）

（ドライブBのディスクを初期化するとともに，MS-DOSのシステムファイル等をこのディスクに

転写せよ．情報教育棟ではできない)

と入力する．(イ)の場合，初期化されたフロッピーディスクはデータの書き込みや読み出ししかできず，このディスク単独ではMS-DOSを起動することができない．(ロ)の場合には，MS-DOSを起動させるために必要な3つのファイル，MSDOS. SYS, IO. SYS, COMMAND. COMを初期化されたディスク上にコピーするので，このディスク単独でもMS-DOSを起動することができる(MSDOS. SYSとIO. SYSは“隠されたファイル”であって，後述のDIRコマンドでは表示されない)．

(6) MS-DOSのコマンドを入力中にタイプミスをした場合は，バックスペースキーで間違えた場所までカーソルを移動させ，入力しなおす．

(7) ディスクの初期化は最初に1度だけ行なえばよい．FORMATコマンドによってディスクを初期化すると，それまでそのディスクに存在していた情報はすべて消去されるので注意を要する．

(b) ファイルの内容表示と印刷

(b)-1 ファイルの内容表示：　TYPE

ファイルの内容をディスプレイに表示するにはTYPEコマンドを使う．

(1) ドライブAのディスクにあるファイル，たとえば「FILE1. PAS」の内容を表示させるには，以下のように入力する．

　　type　a : file1. pas ⏎

　　(ドライブAにあるFILE1. PASというファイルの内容を画面に表示せよ)

ドライブAがカレントドライブである場合には　a：　を省略することができる．

(2) ファイルが大きい場合，画面はスクロールし，ファイルの内容が次々に表示される．ゆっくり内容を見たいときには，次のいずれかの方法をとる．

イ) CTRL-Sを押してスクロールを停止させる．スクロール再開には，何か1文字(たとえばスペース)を入力する．

ロ) 1画面分ずつ区切って表示させるため

　　type　a : file1. pas　|　more ⏎

　とする．1画面分表示したところで停止するので，何か1文字(たとえばスペース)を入力すると次画面面分に移る

(3) このTYPEコマンドや次のPRINTコマンドは，ソースプログラムのファイルや何らかの文書ファイルなど，一般にテキストファイルにだけ使用する．オブジェクトプログラムのファイルやシステムファイル(システムソフトウエアが入っているファイル)に対してこれらのコマンドを実行しても，奇妙な記号が表示されるだけである．

1.1節でも述べたように，われわれ人間が用いる文字列としての情報表現(これを外部表現と呼ぼう)と，計算機内部の情報の表現(内部表現)とは互いに異なっている．このような内部表現と外部表現の間の自動変換が定義されているファイルをテキストファイルという．

(b)-2　ファイルの内容印刷：　PRINT

ファイルの内容をプリンタに印刷するには次のようにする．

(1)　MS-DOS のシステムファイルが入っているディスクのドライブをカレントドライブとする．

(2)　内容を印刷しようとするファイル(FILE2. PAS とする)が入っているディスクのドライブ(いまの場合，ドライブ A とする)を確認する．

(3)　プリンタの電源が ON になっており，用紙がセットされていることを確認する．

(4)　プロンプト　F：¥＞　に対し

F：¥＞print　a：file2. pas ⏎

(システムファイルの PRINT というプログラムを用いて，ドライブ A にある FILE2. PAS の内容をプリンタに印刷せよ)

と入力する．

(c)　**ファイル管理**

(c)-1　ファイル

(1)　われわれがディスク上にソースプログラムやデータや文書など一まとまりの情報を記録しておくとき，これらはすべてそれぞれファイルとして存在する．ファイルは第2章で見たように，TURBO Editor や汎用エディタを通じて，キーボードから文字列を入力して作ることもできるし，3.5 節で見るようにプログラムによってデータファイルを作成したり，その内容を書き換えたりすることもできる．

(2)　ファイルにはそれぞれ名前を付けて識別する．MS-DOS におけるファイル名は，英数字，特殊記号，仮名，漢字などからなる 8 バイト以内(英数字，特殊記号の 1 部など，1 バイトで表わせる文字や記号の場合は 8 文字以内，漢字は 1 字に 2 バイトを要するので許容文字数は減る．すべて漢字なら 4 文字以内)の文字列と，3 バイト以内の拡張子の前にピリオドを付けたものである(2.2 節で述べた TURBO Pascal におけるファイル名の付け方は MS-DOS に準拠したものなのである)．ファイルには，内容や使用目的を反映したわかりやすい名前を付けるのがよい．

(3)　ファイル一覧(ディレクトリ)の表示：　DIR

ディレクトリとは，ディスク内容の一覧であって，ファイル名とその大きさ，および作成または最後に変更された日付などの情報が記録されている．ディレクトリの内容を見るには DIR コマンドを用いる．たとえばドライブ A のディレクトリを表示するには

F：¥＞dir　a：⏎

とする．ドライブ名を省略した場合にはカレントドライブのディレクトリが表示される．

この他

dir　a：　/w ⏎

dir　a：　/p ⏎

などのオプションを指定することもできる．それぞれの機能は一度試してみれば直ちに理解できるであろう．

(4) ファイルの複写： COPY

ファイルのコピーを行なうにはCOPYコマンドを使う．

F：¥＞copy afile. pas a：bfile. pas ⏎

はドライブFにあるAFILE. PASを，ドライブAのディスクにBFILE. PASという名前でコピーする．複写元ドライブがカレントドライブである場合やカレントドライブ内での複写を行なう場合にはドライブ指定を省略することができる．また，新しいファイル名を指定しなければ，COPYコマンドは自動的に送り側のファイル名でコピーを行なう．

(5) ファイルの削除：DEL

不要になったファイルを削除するにはDELコマンドを使用する．

F：¥＞del a：cfile. bak ⏎

はドライブAにあるCFILE. BAKを削除する．カレントドライブ内のファイルの場合にはドライブ指定を省略できる．

(6) MS-DOSの予約ファイル

MS-DOSでは，各種の周辺装置(デバイス)をもファイルとして扱うことができる．そのために，いくつかのファイル名がデバイスファイル名として予約されており，これらのファイル名をディスク上のファイルとして，ユーザが再定義して使うことはできない．予約ファイル名には次のようなものがある(システムによって多少の変動あり)．

AUX 補助入出力(たとえば通信ポート)．

CON コンソールデバイス．キーボードからの入力やディスプレイへの出力．

PRN プリンタデバイス．プリンタへの出力．

NUL ダミー入出力ファイル．

その他

(7) ワイルドカード： 疑問符 ? とアスタリスク ＊

ファイル指定の中に，ワイルドカードと呼ばれる2つの特殊文字，すなわち疑問符(?)とアスタリスク(＊)を使うことにより，MS-DOSコマンドで複数のファイルを1度に取り扱うことができる．

(イ) ファイル名と拡張子の中での疑問符(?)は，その位置に「任意の1文字」があてはまることを意味する．たとえば，DIRコマンドと組み合わせて

DIR REIDAI?. PAS

と入力すると，カレントディレクトリ(本節(c)-3項参照)にあるファイルのうち，拡張子が“.PAS”であり，“REIDAI”という文字列ではじまる7文字のファイル名を持ったすべてのファイルのディレクトリ情報を表示する．このコマンドで表示されるファイルは，たとえば次のようなものである．

REIDAI1.PAS

REIDAI2.PAS

REIDAIA.PAS

REIDAIX.PAS

(ロ)　ファイル名と拡張子の中でのアスタリスク(*)は，その位置の任意の複数個の文字が可変であることを意味する．たとえば

DIR　*.PAS

は，カレントディレクトリにあるファイルのうち，拡張子が“.PAS”であるファイルのディレクトリ情報を表示する．

例　REIDAI.PAS

KADAIX.PAS

AFILE.PAS

(c)-2 ディスクドライブ

(1) われわれが多数のファイルを保有するようになると，これらのファイル群を何らかの方法で整理・統括することができれば便利である．MS-DOS では，ファイル群をグループ化し組織化する方法として，ドライブによるグループ化と，ディレクトリによる組織化を用いることができる．

(2) ファイル群グループ化の最大の単位はディスクドライブによる区別である．ディスクドライブは英字1文字のドライブ名で指定する．複数個のディスクドライブを備えたシステムでは，各ドライブを指定・利用して，多様なファイル操作が可能となる．

(3) ドライブ名を省略したとき MS-DOS が入出力を行なうドライブをカレントドライブという．カレントドライブは MS-DOS のプロンプト中に表示される．カレントドライブの変更法についてはすでに述べたが

d_1：¥＞d_2：⏎

はカレントドライブを d_1 から d_2 に変更する．

(c)-3 ディレクトリ

(1) 同一ディスク上でファイル群を分類整理するには，図2.5に示すような，階層ディレクトリ構造を利用すると便利である．すなわち，MS-DOS においては，いくつかのファイルやディレクトリをまとめて1つのディレクトリとして登録することができる．この場合，ディレクトリにもファイル名と同様な方法で名前を付けてディレクトリ名とする．換言すれば，1つのディレクトリの中にいくつかのファイルや別の(サブ)ディレクトリを持つことができ，その(サブ)ディレクトリの中にさらに別のファイルや(サブ)ディレクトリを持つ……という構成にできるのである．

(2) 各ドライブには，1番上の階層のディレクトリとしてルートディレクトリがある．このルートディレクトリは，ディスクのフォーマットを行なったときに自動的に作られるディ

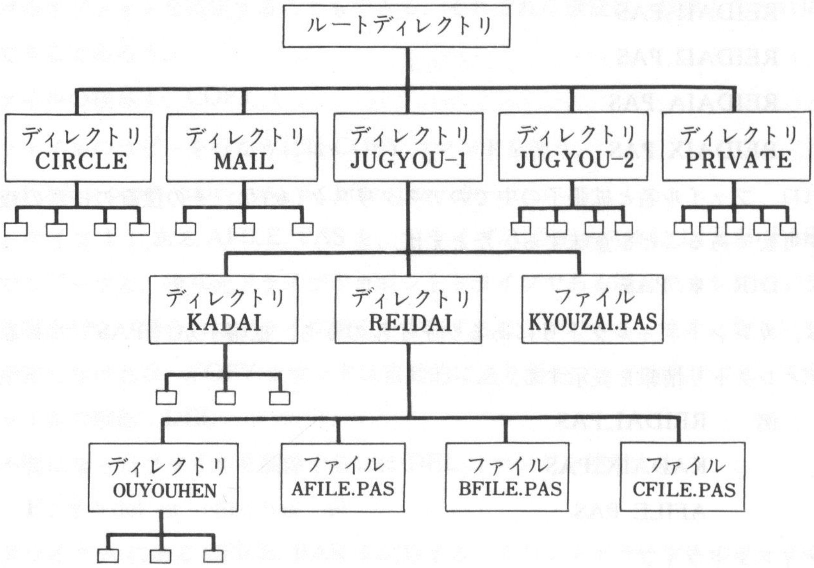


図 2.5　階層ディレクトリ構造の例

レクトリである．ユーザは必要に応じてルートディレクトリの下に，(サブ)ディレクトリを作ることができる．(サブ)ディレクトリを作るにはMD (Make Directory) コマンドを使う．

A：¥>md jugyou1 ↵

は，ルートディレクトリの下にJUGYOU1なるディレクトリを作る．

(3) すべてのファイルは，ルートディレクトリからそのファイルに至るディレクトリ名の連鎖とそのファイル名自身とによって一意に指定される．これをそのファイルの完全パス名と呼ぶ．たとえば

A：¥JUGYOU1¥REIDAI¥AFILE.PAS

によってドライブAのルートディレクトリに属し，JUGYOU1というディレクトリに属し，その中でもREIDAIというディレクトリに属するAFILE. PASというファイルが指定される．このように「ディレクトリ……に属する」ということを「……¥」で表わす．ルートディレクトリを表わすには単に通貨記号 ¥ のみを用いる．

(4) 各ドライブごとに，当面の処理の中心的対象となるディレクトリを1つ指定することができる．これをカレントディレクトリと呼ぶ．MS-DOS起動時のカレントディレクトリはルートディレクトリである．上の(2)で述べたMKDIRコマンドはカレントディレクトリの下に新しいディレクトリを作るのである．また，カレントディレクトリの移動にはCD (Change Directory) コマンドを用いる．たとえば

A>cd　¥jugyou1 ↵

によって，ルートディレクトリの中のJUGYOU1というディレクトリをカレントディレ

クトリとして指定できる．なお，パス名を省略して CD コマンドだけを入力すると，その時点でのカレントディレクトリが表示される．

(5) カレントディレクトリは略号ピリオド「.」で表わされる．またカレントディレクトリの親ディレクトリは略号「..」で表わされる．

(6) カレントディレクトリ内のファイルやディレクトリを指定する場合には，完全パス名を全部指定するかわりに，単にそのファイル名やディレクトリ名だけを指定すればよい．たとえば先の AFILE. PAS なるファイルの指定は，完全パス名による場合

A：¥JUGYOU1¥REIDAI¥AFILE.PAS

であったが，カレントディレクトリが

A：¥JUGYOU1¥REIDAI

である場合には，単に AFILE. PAS で足りる．このようにカレントディレクトリを基点とするファイルの指定を相対パス名と呼ぶ．

(d)　日本語入力機能

わが国で販売・使用されている MS-DOS は多くの場合日本語 MS-DOS であって，日本語の入力機能を備えている．これらにおいては ATOK や OAK など，本来の MS-DOS システムとユーザプログラムの間に位置して日本語入力を処理する，いわゆる“フロントプロセッサ”タイプのソフトウエアが組み込まれている．日本語入力のためのキー操作はパソコンの機種によって微妙に異なるのでここでは詳述を避けるが，基本的には次の手順による．

(1) まずパーソナル・コンピュータを日本語入力モードにする．

NEC の PC-9800 シリーズでは　CTRL キーと XFER キーを同時に押す．

IBM の 5550 シリーズでは　漢字キーを押す．

富士通の FMR では　かな漢字キーを押す．

(2) 日本語入力フロントプロセッサは通常さまざまな入力モードを備えているが，一般的には，「ローマ字」による入力を自動的に「ひらかな」に変換するモードを選ぶ．画面の表示が，「R (ローマ字入力)」，「ひら」または「かな」，「(半角でなく)全角」などとなるようにファンクションキー等を用いて調整する．

(3) キーボードから入力した日本語は自動的にひらかなに変換して表示されるので，必要に応じて漢字変換キー(上の順に，XFER キー，変換キー，変換キー)を押せば，漢字・かな混じり文に変換される．詳しくは各システムのマニュアルを参照されたい．

第3章 Pascal文法の基礎

3.1 データの型と式

(a) 数を表すデータの型

第1章でPascalの基本的なデータの型を紹介した．TURBO Pascal ver. 5では数を表すデータの型が次のように何種類もある．この中で◎は標準的に使う型である．

定義済み整数型

型	範囲	形式
shortint	−128～127	符号付き8ビット
◎integer	−32768～32767	符号付き16ビット
longint	−2147483648～2147483647	符号付き32ビット
byte	0～255	符号なし8ビット
word	0～65535	符号なし16ビット

定義済み実数型の範囲と桁数(10進)

型	範囲	桁数	バイト数
◎real	2.9×10^{-39} ～1.7×10^{38}	11～12	6
single	1.5×10^{-45} ～3.4×10^{38}	7～ 8	4
double	5.0×10^{-324} ～1.7×10^{308}	11～12	8
extended	3.4×10^{-4932}～1.1×10^{4932}	19～20	10
comp	$-2^{63}+1$ ～$2^{63}-1$ (-9.2×10^{-18}～9.2×10^{18})	19～20	8

整数型は整数値を表現するデータ型である．これに対し，実数型は浮動小数を表現できる．整数型の代表はinteger，実数型の代表はrealであり，普通はこれらの型を用いる．数値を扱う場合は，必要に応じて整数型か実数型かまた数値の範囲によってデータの型を選ばねばならない．integer型の最大値は32767なので，例えば日本の人口とか大きい金額を表すにはlongint型が望ましいことがわかる．

(b) 整数・実数計算

第1章で整数の四則演算を行ったので，整数・実数が混ざった簡単な計算を行ってみる．

[例題 3.1] 所要時間の計算

平均時速と目的地までの距離を入力し，所要時間を計算する．

プログラム 3.1

```
{  *** 所要時間の計算 ***  }
program ex2_1(input,output);
var speed, dis, time : real;
    hour, min : integer;

begin
  write('速度(km/h) ? ');
  readln(speed);
  write('距離(km) ? ');
  readln(dis);
  time := dis / speed;
  hour := trunc(time);
  min := round(60*(time - hour));
  write(time:6:4,' 時間 = ');
  writeln(hour,' 時間 ',min,' 分')
end.
```

◆実行例

```
速度(km/h) ? 43
距離(km) ? 123
2.8605 時間 = 2 時間 52 分
```

所要時間は t 時間という形と h 時間 m 分という形で表示する．速度 s，距離 d，時間 t は実数型，h，m は整数型で表す．所要時間 t は，t = d/s で計算できる．t 時間を h 時間 m 分に変換するには，h は t の整数部分であり，m は t の小数部分を 60 倍して四捨五入する．

使用するすべての変数は前もって宣言する．宣言を行う場所は実行部分の前である．変数宣言は必ず次のような形式になっている．

> var <u>変数リスト：型名；</u>
>
> 　　複数の型を用いるときはこの部分を繰り返す

整数型変数と実数型変数を宣言すると，例えば，

var age : integer ;

weight, height : real ;

のようになる．変数リストとは，変数が1つのときはそのまま，変数が複数個のときは','で区切って並べる．型名と言うのは，整数型のときは integer(他に longint, byte など)，実数型のときは real(他に double など)という名前を用いる．

計算した結果を変数に代入するには，次のような代入文を用いる．

変数：＝式

：＝の前後を空けることはできるが，：＝の間を空けることはできない．式とは，変数や定数(この場合オペランドという)を演算子で結び付けたものである．一般の数式と同じように書くが違う点もいくつかある．

第1章で整数型の加減乗除と剰余の計算で＋，－，＊，div, mod の演算子を扱ったが，div, mod は整数型だけに使える演算子で，実数型の加減乗除には＋－＊/の記号を用いる．実数型の剰余計算はない．また / の演算は整数型にも使えるが，結果は実数型となる．すなわち，

5 div 3　と　5/3

では結果が異なる．前者は整数型の1となり，後者は実数型の1.6666……になる．

演算子の前後のオペランドの型が異なる場合は，実数型に変換されて計算される．

演算子	意味	優先度	例	注意事項
＋	加算	低	a＋b　a＋1.23	
－	減算	低	a－b　3.0－b	単項演算もある
＊	乗算	高	a＊b　132.5＊b	
/	除算	高	a/b　a/23.5	結果はつねに実数型
div	整数型除算	高	n div m	整数型のみ
mod	整数型剰余	高	i mod 2	整数型のみ

オペランドはいくらでもつなげることができる．一般の式と異なり乗算記号は省略することはできない．

a＊b＋c－d/e＊f－g

この場合，＋－より＊/の方が優先度が高く，先に演算を行う．同じ優先度のときは，左側から順に演算が行われる．例えば a/b/c は a を b で割ってさらに c で割る．a/b＊c は a を b で割った値に c を掛ける．

優先順位を変えるときは，一般の式のように(　)を用いる．例えば，次のようになる．

(a＋b)＊c　　((a＊x＋b)＊x＋c)＊x＋d

ただし{　}や[　]は用いずに，(　)を重ねて用いる．

演算子のないオペランドだけでも'式'として認められる．したがって式の特別な場合は変数だけ，定数だけになる．このような考え方をすると次のような例もすべて上の　変数：＝式　という代入文で表すことができる．

代入の形	例	意味
変数：＝定数	a：＝1.23	変数に特定の値を代入する
変数：＝変数	b：＝a	変数の値を他の変数に移す
変数：＝−変数	c：＝−b	変数の値の符号を反転して変数に移す
変数：＝変数＋定数	n：＝n+1	変数nの値を1つ増やす(例の場合)

減算については次の例のように，1つのオペランドに対し，単独で用いることができる．この減算を単項演算といい，最も優先度の高い演算子として扱われる．

a：＝b/−c

なお，べき乗の演算はないのでaの2乗，xの3乗はa*a，x*x*xなどと書かなければならない(aのb乗をexp(b*ln(a))で計算することもできる)．さらに定数については，12E3(＝12,000)，1.83e−4(＝0.000183)のように指数表現もできる(実数型)．この他$12Fなどの16進法も使える(整数型)．$は16進法表記を意味し，$\$12F = 1\times16^2+2\times16+15$である．

整数型と実数型の混合演算で注意することは，演算結果を変数に代入する場合である．実数型の変数にはそのまま代入できるが，整数型変数に実数型の値を代入するには，切り捨てか四捨五入の型変換関数を用いなければならない．

trunc(x)　　実数(式)xを切り捨てて整数型に変換する

round(x)　　実数(式)xを四捨五入して整数型に変換する

truncと似たような関数にintがあるが，intは実数値の整数部分を取り出してその値を実数型で返す関数である．

例：　a：＝round(1.5)；

　　　i：＝trunc(a*b+1.4)；

　　　j：＝trunc(c+3.5)mod round(a/b)；

> ◆オーバーフローについて
> 計算途中で，表現できる数値の範囲を越えてしまうことがある．TURBO Pascalでは，整数型の場合，0の除算を除いてエラーメッセージを出さないでそのまま計算を続ける．したがって正の整数の足し算や掛け算でオーバーフローして数値がマイナスになることがある．実数型の計算でオーバーフローをすると実行を中断し，エラーメッセージを出してその箇所を示してくれる．

(c)　数の入出力

計算だけを行うプログラムでも，数値を入力したり，結果を画面に表示しなければならない．そのような数の入出力について説明する．

変数にキーボードから読み込んだ値を代入するには，

read(変数リスト)　または　readln(変数リスト)

という手続き文を用いる．read と readln の違いはない*．

1つの変数の値を読み込むには，

```
read(a);
```

複数の変数の値を読み込むには，

```
read(a, b, c);
```

のように記述する．(　)の中の a, b, c のことを引数(ひきすう)という．プログラムで read, readln 文が実行されると，キーボードから数値を入力するわけであるが，1つの変数に対しては数値を打って⏎キー，複数の変数に対しては数値をスペースキーまたは TAB キーで区切って打ち込んだ後に⏎キーを入力する．

read, readln の実行において，何の値を入力するのかわからなくなることがあるので，次のように write 文とペアにして用いるとよい．

```
write('age ? '); readln(age);
write(' データ数？'); readln(n);
write(' 身長と体重？'); readln(height, weight);
```

数値の値を画面に表示するには，write, writeln を用いる．read 文では，引数は必ず変数であったが，write 文の場合は式や文字列が使える．write は表示後に改行しないのに対し，writeln は改行を行う．

write(パラメータリスト)　または　writeln(パラメータリスト)

パラメータリストは1つのパラメータまたは複数のパラメータを，(コンマ)で区切ったものである．パラメータが　'文字列'　であれば，その文字列を画面に表示するし，パラメータが変数であれば変数の値を，定数であればその値を10進数表示する．

整数型変数 i, j, k の値を表示する場合，たとえば，i = 12，j = 3，k = 456 のとき，

```
writeln(i, j, k);
```

では画面上で 123456 となってしまうので，次のように桁幅を指定するか，説明文を付け加えると表示がわかりやすくなる．パラメータの次にくる：7 は7桁の桁幅に右に寄せて表示されるので見やすくなり，表などを作成する場合に便利である．また意味のある言葉，説明文をつけてもよい．

```
writeln(i : 7, j : 7, k : 7);
writeln('i = ', i, '  j = ', j, '  k = ', k);
```

---

*　ディスクファイルから読み込む場合は異なる．また，TURBO Pascal ver. 3 ではデータを読み込んだ後に改行を行うという違いがあった．

writeln(' 最小値 = ', i, ' 最大値= ', j);

実数型については，変数 x の値が x = 1234.56789 のとき，writeln(x); が実行されると，

1.23456789000000E+0003

と画面上に表示される*．この形式は指数形式といい，E の前の数値を仮数部，E の後ろの数値を指数部という．仮数に 10 の指数乗を掛けた値が結果となる．1 より小さい数値については指数部が－となる．

実数型の場合も桁幅を指定することができ，x = 12.3456789 について 7～17 の桁幅を指定すると次のようになる(左側)．

writeln(x: 7)	1.2E+0001	writeln(x:10: 0)	12
writeln(x: 8)	1.2E+0001	writeln(x:10: 1)	12.3
writeln(x: 9)	1.2E+0001	writeln(x:10: 2)	12.35
writeln(x:10)	1.2E+0001	writeln(x:10: 3)	12.346
writeln(x:11)	1.23E+0001	writeln(x:10: 4)	12.3457
writeln(x:12)	1.235E+0001	writeln(x:10: 5)	12.34568
writeln(x:13)	1.2346E+0001	writeln(x:10: 6)	12.345679
writeln(x:14)	1.23457E+0001	writeln(x:10: 7)	12.3456789
writeln(x:15)	1.234568E+0001	writeln(x:10: 8)	12.34567890
writeln(x:16)	1.2345679E+0001	writeln(x:10: 9)	12.345678900
writeln(x:17)	1.23456789E+0001	writeln(x:10:10)	12.3456789000

また実数型変数については，指数形式でない固定小数表示をすることができ，その場合，全体の桁幅と小数点以下の桁幅を：で区切って指定する．同じ x について 10 桁幅で小数点以下 0～10 桁を指定した場合の表示は上の右のようになる．

これらをまとめると write, writeln 文のパラメータ x は，次のようになる．

x	x が実数型の場合は指数形式
x : f	f 桁の幅に右詰め，実数型のときは指数形式
x : f_1 : f_2	x は実数型，f_1 は全体，f_2 は小数点以下の桁数

x は式，変数，定数，' 文字列 '

f, f_1, f_2 は整数型の式(変数，定数)

なお f, f1, f2 は整数型の式であることを考えると，次のような例も考えられる．

例： writeln(i : j+2);

writeln(' 平均= ', (a+b+c)/3.0 : 8 : 3);

* コンパイラオプションとして数値演算プロセッサの使用が指示されているため，表示は extended 型となる．

```
writeln('data= ', b/c : j+2 : 3*k);
writeln(' 右側 ' : 60);
writeln(123 : 10,1.23 : 10 : 4);
```

(d)　標準関数

一般の計算によく用いる三角関数，指数・対数関数などはシステム側で用意されている(標準関数という)．算術計算用の標準関数には次のようなものがある．

関数名	引数	結果の型	意味
abs	整数型	整数型	絶対値
	実数型	実数型	
sqr	整数型	整数型	2乗
	実数型	実数型	
sin	整/実数型	実数型	正弦関数
cos	整/実数型	実数型	余弦関数
exp	整/実数型	実数型	指数関数
ln	整/実数型	実数型	自然対数
sqrt	整/実数型	実数型	平方根
arctan	整/実数型	実数型	逆正接関数
int	整/実数型	実数型	実数の整数部分
frac	整/実数型	実数型	実数の小数部分
pi	なし	実数型	円周率 π を返す

たとえば実数型変数 a の絶対値は　abs(a)と表し，a のことを引数(ひきすう)という．abs(a)も実数型になるので，別の変数に代入してもよいし，式に組み込んでもよいし，write 文の中でつかってもよい．

例：
```
b := abs(a);
c := (abs(a) + 2.0) /3.0;
writeln(' 絶対値= ', abs(a));
d := sqrt(sqr(b) - 4.0*a*c);
x := sin(abs(x+y) + z) + cos(x/6.0);
```

引数は実数型でも整数型でもよく，abs と sqr を除いては結果は実数型になる．三角関数の sin，cos の引数や arctan の結果の角度の単位はラジアンである．sin や cos はあっても tan はないので，次の例題のように自分で計算を組まなければならない．

[例題 3.2]　建物の高さ

建物から d(m)離れた点から建物の最上部を見上げた時の角度が $\theta°$ であった．建物の高さ h (= d tan θ とする)を求める．

プログラム 3.2

```
{ *** 建物の高さ *** }
program prog3_2(input,output);
const deg_rad = 3.14159265/180;
var d, t, h : real;

begin
  write('距離は(m) ? '); readln(d);
  write('角度は(deg) ? '); readln(t);
  t := t * deg_rad;
  h := d * sin(t) / cos(t);
  writeln('高さは ',h:5:3,'m です.')
end.
```

◆実行例

```
距離は(m) ? 25
角度は(deg) ? 40
高さは 20.977m です.
```

定数に名前をつけて前もって準備することができる．これは定数定義部と呼ばれる部分である．定数の書き方により，整数型や実数型などの型が決まる．たとえばa＝1； であればaは整数型，a＝1.0； であればaは実数型になる．定数の値は，1.23e−4のように指数表現を用いることもできるし，四則演算を用いることもできる．

定数定義には次のように2種類あり，このプログラムの場合は型なし定数定義である．TURBO Pascal ver. 5の場合，定数の記述に式が使える(関数などは使えない)．

型なし定数定義	const 名前＝式；....	定数
型付き定数定義	const 名前：型＝式；....	変数の初期設定

例： const a = 1 ; b = 1.0 ;
c = 1.23e−4 ; d = 1E3 ;
e : integer = 4 * 123 ;
f : real = 0.582 ;
weeks = 365 div 7 ;

型なし定数定義の場合，a, b, c, d, weeksは普通の変数とは区別され，値を代入することはできない．型付き定数定義では，e, fとも初期設定を除いては普通の変数と同じように使用できる．

◆計算精度について

整数型の途中の計算は longint 型で行われる．したがって，

```
writeln(round(sqrt(2)*1e8))
```

を実行すると，正しい値が表示されるが，これを

```
i:=round(sqrt(2)*1e8)
```

と integer 型変数に代入してから，表示するとでたらめな値となる．実数型の途中計算も，real 型を extended 型に変換してから計算する（数値演算プロセッサの指定がある場合）．計算のオーバーフローや精度には十分注意したい．

(e)　練習問題

[1-1]　2つの抵抗値 a，b を入力し，並列に接続したときの合成抵抗 c を計算しなさい．ただし，

$$\frac{1}{c}=\frac{1}{a}+\frac{1}{b}$$

[1-2]　価格 x (longint 型) を入力して消費税 (3%) と税込み金額を表示しなさい．

[1-3]　三角形の辺の長さ a，b，c を入力してヘロンの公式で面積 s を求めなさい．$2p=a+b+c$ とすると，$s^2=p(p-a)(p-b)(p-c)$ である．

[1-4]　ある金額を入力し，100円，10円，1円玉だけで支払うとき，それぞれ何枚必要か（できるだけ少なく）を表示しなさい．

[1-5]　平面上の点 (x, y) の座標値を入力して ($x>0$ とする)，原点との距離 $\sqrt{(x^2+y^2)}$ と，x 軸となす角（度）を計算しなさい．

[1-6]　2数 a, b の値を入力して，相加平均と相乗平均を計算しなさい．a, b ともに正とする．

[1-7]　実数 x の値を入力し，小数点以下第2桁目を四捨五入および切り捨てて，表示しなさい．

[1-8]　x の値を入力して次の関数値を計算しなさい．a, b, c, d は定数宣言で与え，乗算の回数をなるべく少なくしなさい．$f(x)=ax^3+bx^2+cx+d$

ヒント：((a*x+b)*x+c)*x+d という式を用いる．

[1-9]　2つの複素数 a+bi，c+di の実部・虚部 a, b, c, d を入力し，四則演算結果の実部・虚部を表示しなさい．

[1-10]　2数 a, b の値を入力して四則演算の結果を表示しなさい．ただし exp と ln を用いて，加減算については乗除算で，乗除算については加減算で結果を求めなさい．

3.2　条件判定

(a)　if 文

ある条件を満たしたときに，特定の処理を行うのが if 文の役目である．日常会話で次のような例がある．

A：もし「雨」ならば「洗濯物を取り込む」

B：もし「雨」ならば「映画に行く」そうでなければ「ハイキングに行く」

これらに対応して if 文の書式は次のように2通りある．

A (if-then)	:	if 論理式 then 文
B (if-then-else)	:	if 論理式 then 文$_1$ else 文$_2$

論理式は，結果が論理型になる式や論理型変数のことで，普通は整数や実数の大小比較などが論理式になる．大小比較は2つの式や変数，定数を ＜ ＞ ＝ などの記号で結んだものである．

```
if a < 0 then a := -a
```

このif文では，a<0の部分が論理式である．論理式の結果は，true (真) または false (偽) のどちらかになる．式が成立していれば true，それ以外は false になる．このif文が実行されたときのaの値が−1とか−2の場合は true になり，後ろの代入文が実行される．この例では結果として $|a|$ を求めたことになる．A (if-then) の形式では，論理式が true である場合に限り文を実行する．B (if-then-else) の形式では，論理式が true である場合に文$_1$を実行し，論理式が false の場合には文$_2$を実行する．Bの形式では，else の前の文に ; を付けてはならない．

```
if b*b >= 4*a*c then writeln('実根あり')
                 else writeln('実根なし')
```

このif文では，b^2の値が4ac以上の場合は論理式が true になり，'実根あり' と表示され，それ以外の場合は '実根なし' と表示される．

大小比較には次のような6種類があり，その結果は論理型になる．これらの比較を関係演算子という．

A＝B	：AはBに等しい	A＜＞B	：AはBに等しくない
A＞B	：AはBより大きい	A＞＝B	：AはB以上である
A＜B	：AはBより小さい	A＜＝B	：AはB以下である

A，Bは整数型または実数型の式(変数，定数)などである．普通は，整数型どうし，実数型どうしを比較するが，整数型と実数型を比較した場合は実数型に変換されて比較される．なお，A＞Bは，B＜Aと同じであり，A＞＝Bは，B＜＝Aと同じことを表している．

条件が成立した場合に複数の文を実行したいときは，文を複合文にする．つまりそれらの文を begin と end で囲む．同じように，条件が不成立の場合に複数の文を実行したいときは else 以下を複合文にする．このときも else の前の end に ; を付けてはならない．

```
if i mod 2 = 0 then begin
  writeln(i, '=偶数');
  i := i div 2
end else begin
  writeln(i, '=奇数');
  i := 3*i+1
end
```

[例題 3.3]　座標の象限

平面の座標値 xy を入力して，第何象限かを判定する．原点，xy 軸上は考えなくてよい．

プログラム 3.3

```
{  *** 座標の象限 ***  }
program prog3_3(input,output);
var x, y : real;
    i : integer;
begin
  write('座標の x y ?');
  readln(x,y);
  if x>0 then i:=1 else i:=2;
  if y<0 then i:=5-i;
  writeln('第 ',i,' 象限です.')
end.
```

普通に考えると，

if x ＞ 0 then

　　if y ＞ 0 then　i：＝1 else　i：＝4

else

　　if y ＞ 0 then　i：＝2 else　i：＝3

となるが，上のプログラム 3.3 のようにもできる．

◆実数値の比較について

計算で得られた実数値を，

if　x＝y　then....

のように比較する場合は十分注意したい．計算には誤差が含まれるので，場合により，

if abs(x−y) ＜eps then....

のようにしなければならない．eps の値は計算の種類や型(real, extended など)の違いで変わるが 1.e-8～1.e-16 程度の値が用いられる．実数の四則演算や関数の計算には誤差が含まれるということに注意したい．

(b)　論理演算

上の例題で第1象限だけを表す場合，

if　x ＞ 0　then　if　y ＞ 0　then　i：＝1

という2重の if 文になるが，これは x ＞ 0 という条件と y ＞ 0 という条件が両方同時に成立する場合である．言い換えると，x ＞ 0 かつ y ＞ 0 となる．日常会話でも，A と B の件が両方生じた場合とか，A または B どちらかが生じた場合など，論理的な場合分けを行なうことがよくある．

もし「雨」または「くもり」ならば「花は開かない」

演算子	例	意　味
and	(0 < x) and (x <= 5)	論理積，両者が true に限り true
or	(i = 2) or (i = 3)	論理和，どちらかが true のとき true
xor	(i < > 0) xor (j = 0)	排他論理和，両者の値が異なるとき true
not	not (x < 5)	反転(true→false, false→true)

もし「雨」かつ「強風」ならば「雨漏りがする」

プログラムを組む場合も，このような場合分けは頻繁に起こり，そのための論理演算が用意されている．

このような論理演算には，次のような論理積，論理和，排他論理和，否定がある．

論理積や論理和の演算の場合，true → 1，false → 0 と考え，0 と 1 の積・和を考えると理解しやすくなる．排他論理和(exclusive or)の演算は前後の論理式の値が異なると true，同じであると false になる．

二つの論理式を and でつなぐと二つとも true のとき全体の値が true になる．たとえば，

(0 <= a)　and　(a < 5)

この論理式では，たとえば a = 3 のときは and の前後の両方の論理式が true になり，全体が true になる．a = −1 や a = 6 の場合は一方が true でもう一方が false であるから，全体では false になる．つまりこの論理式は 0 以上 5 未満の数を示すことになる．同じようにこれ以外の範囲の数を示すには，

(a < 0)　or　(5 <= a)

となる．なお，これらの記述で，and, or の方が演算の優先順位が高いため，(　)は省略できないので注意すること．

ところで，① if a then A という文で，文 A が if b then B else C という if 文であるとすると全体では，

if a then if b then B else C

という文になる．一方，② if a then A else C という文で，文 A が if b then B という文であるとすると全体では，

if a then if b then B else C

と ① の例とまったく同じになる．このような場合，else に対応する if はその直前の if ということになっているので，① の形であるとみなされる．② の例のようにしたい場合は if b then B を be-

◆ショートサーキット

if A and B then.... という if 文では A = false のとき A and B 全体は必ず false になる．このとき B が true か false かを調べないのが short circuit で，調べる処理を complete という．TURBO Pascal ではコンパイル時のオプションでどちらにするかを選択できる．

if A or B then.... で A = true のときも同じである．

gin　end で括る必要がある.

if a then begin if b then B end else C

(c) case 文

if 文を使って幾通りかの場合分けをすることができる．たとえば郵便物の重さが 25 g 以下のときの料金は 62 円，それ以上は 72 円であるから，重さを入力して料金を決めるという 2 通りの場合分けは次のようになる．

```
if weight <= 25 then charge := 62
              else charge := 72;
```

もっと細かい場合わけを行う場合は(定形外 1 kg 以下の郵便物)，

```
if weight <= 50 then charge := 120
  else if weight <= 100 then charge := 175
    else if weight <= 250 then charge := 250
      else if weight <= 500 then charge := 360
        else charge := 670
```

となり，if～else if～else if～……else という構文で細かい場合分けが行えることがわかる．

場合分けで，整数値の様々な値に対応して異なる処理を行なう場合は，case 文が役に立つ．case 文の書式は，次の通りである．

```
case 整数式 of
  case ラベル, ..., case ラベル : 文;
  ....
  case ラベル, ..., case ラベル : 文;
  else 文; ... 文;
end
```

case ラベルは，

① 定数　　　　　　または

② 定数$_1$.. 定数$_2$　　(定数$_1$ ≦ 定数$_2$)

である．case 直後の整数式(結果が整数型になる式)の値が case ラベルに対応するとき，つまり ① の場合は定数の値に等しいとき，② の場合は，定数$_1$ 以上定数$_2$ 以下のとき，その次の文を実行し，end へ進む(case 文を終了する)．case ラベルに対応する文が複数のときは複文(begin　end で囲む)にする．

すべての case ラベルに対応するものがないときは else の次の文の集まりを実行する．ただし「else 文 ; ... 文 ;」の部分を省略することもある．

例：

case month of　　　(month は整数型の変数とする)

　1,3,5,7.. 8,10,12：days：= 31；

　4,6,9,11：days：= 30；

　2：days：= 28；

end

[例題 3.4]　点数に対する成績

ある大学では，80 点以上，65 点以上，50 点以上，それ未満についてそれぞれ，A, B, C, D の成績がつく．点数を入力して成績を表示する．

プログラム 3.4

```
{  *** 点数に対する成績 ***  }
program prog3_4(input,output);
var i : integer;
begin
  write('点数 ? '); readln(i);
  write('成績は ');
  case i of
    0..49 : write('D');
   50..64 : write('C');
   65..79 : write('B');
   80..100: write('A');
   else write('入力エラー')
  end;
  writeln(' です.')
end.
```

(d)　論理型変数

ところで，整数や実数同様，論理型変数がある．論理型は，true(真)または false(偽)のどちらかの値をとる．型名は，boolean である．

　　var bool：boolean;

次のように定数や大小比較の結果を代入することができる．

　　bool：= false；

　　bool：= a > 5；

　　bool：= not(a <> b)；

　　bool：= b∗b−4.0∗a∗c = 0.0；

論理変数の論理演算は，A and B，A or B，not A のようにその結果も論理型となる．TURBO

Pascal では論理式の値を write 文により出力すると，値にしたがって TRUE または FALSE と表示される．ただし入力することはできない．また ord という関数により，論理式の値を 0(false) または 1(true) の整数型に変換できる．

◆整数の内部表現と bit 演算

整数型についても，論理演算が定義されている．整数型の内部表現は 2 進数で表され，論理演算には次のような演算が用意されている．これらは

i and j	i と j の対応するビットどうしの論理積
i or j	i と j の対応するビットどうしの論理和
i xor j	i と j の対応するビットどうしの排他論理和
not i	i の各ビットの 0 と 1 を交換
i shl j	i を j ビット左シフト(左にずらす)
i shr j	i を j ビット右シフト(右にずらす)

例　i ＝ 0101001101001111
　　j ＝ 1010111100110100

i and j ＝ 0000001100000100	not i ＝ 1010110010110000
i or j ＝ 1111111101111111	i shl 3 ＝ 1001101001111000
i xor j ＝ 1111110001111011	i shr 3 ＝ 0000101001101001

式に一般の数値計算の演算子や関係演算子，論理演算子が混ざっている場合の優先順位は次のようになる．

1) 単項演算子：−
2) not 演算子：not
3) 乗法演算子：＊, /, div, mod, and, shl, shr
4) 加法演算子：＋, −, or, xor
5) 関係演算子：$=, <>, <, >, <=, >=$

なお論理値の大小関係は，論理演算でやったように false ＝ 0，true ＝ 1 と考えてよく，false $<$ true となっている．したがって，

　　if　$(x = 0) = (y <> 0)$ ‥‥
　　if　$(x < 0) < (y < 0)$ ‥‥

のように論理値の大小比較の記述もできる．論理値どうしを $<>$ で比較することは exor に等しいことがわかる．また上の表の優先順位より，

　　A or not B and C

という論理式では，①B の否定，②C との論理積，③A との論理和の順で演算が行われる．

(e) 練習問題

[2-1] 通常郵便で送れる箱の大きさは幅 a, 奥行き b, 高さ c がすべて 60 cm 以下で合計が 90 cm 以下である. a, b, c を入力して通常郵便かそれ以外かを表示しなさい.

[2-2] 身長, 体重を入力して肥満か標準か痩せ型かを示しなさい. 体重が(身長−100) ∗0.9±10%の範囲であれば標準とみなす.

[2-3] 3 数 a, b, c を入力して最大値を示しなさい.

[2-4] 4 数 a, b, c, d を入力して最大値を示しなさい.

[2-5] 平面座標値 x, y を入力して x 軸となす角度 t°(0 ≦ t < 360) を計算しなさい.

[2-6] 3 辺の長さ a, b, c を入力して三角形が成立するかどうかを判定しなさい. a, b, c は正とする.

ヒント：|b−c|, b+c, a の大小関係は？

[2-7] 西暦年(1868 以上)を入力して, 明治, 大正, 昭和, 平成何年かを表示しなさい. 明治, 大正, 昭和, 平成元年は, それぞれ 1868, 1912, 1926, 1989 年である. 元年を優先するが, 元年の表示は 1 年でもよい.

[2-8] 西暦年 y を入力して閏年かどうかの判定をしなさい. 4 で割り切れるが 100 で割り切れず, ただし 400 の倍数であれば閏年である.

[2-9] 月と日を入力してその星座を表示しなさい.

3/21～4/20	牡羊座	7/23～ 8/22	獅子座	11/23～12/21	射手座
4/21～5/21	牡牛座	8/23～ 9/23	乙女座	12/22～ 1/20	山羊座
5/22～6/21	双子座	9/24～10/23	天秤座	1/21～ 2/18	水瓶座
6/22～7/22	蟹　座	10/24～11/22	蠍　座	2/19～ 3/20	魚　座

[2-10] 3 つの数 a, b, c を入力してメジアン(中央値)を示しなさい.

3.3 繰り返し処理

Pascal には文の反復を行う 3 種類の繰り返し文がある. 繰り返し回数がわかっている場合には for 文, 条件によって繰り返しを行う場合には while や repeat 文が利用できる.

■単純繰り返し	for 文
■条件成立繰り返し	while 文
■条件不成立繰り返し	repeat 文

(a) 単純繰り返し(for 文)

ある処理をあらかじめ決まった回数だけ繰り返し行なうために用いるのが for 文である. for 文には次の 2 種類がある.

for 制御変数：=式$_1$ to 式$_2$ do 文

for 制御変数：=式$_1$ downto 式$_2$ do 文

制御変数，式$_1$，式$_2$には整数型を用いることが多い．上の形式は，制御変数の値が式$_1$の値から式$_2$の値まで順に1つずつ増加しながら，doの後ろの文が繰り返し実行される．普通，式$_1$の値＜式$_2$の値であり，式$_1$の値＞式$_2$の値の場合は文は一度も実行されない．式$_1$の値＝式$_2$の値の場合は一度だけ実行されることになる．

toの代わりにdowntoを使えば，制御変数値が減少する向きの反復も指定できる．反復されるのは1個の文であり反復実行すべき文が複数のときは，begin　endではさんで複合文とする．制御変数の型は整数型（正確には実数型以外のすべての順序型）であるが，制御変数，式$_1$の値，式$_2$の値は同じ型でなければならない．

for文の文例集

① 1～nの和
```
s := 0;
for i := 1 to n do
  s := s+i
```
② 式$_1$，式$_2$が式
```
for i := j+1 to 2*j do
  a := a+i
```
③ 繰り返しが複合文
```
for i := 1 to n do begin
  writeln('i = ', i);
  k := k+i
end
```
④ 多重の繰り返し
```
for i := 1 to n do
  for j := 1 to m do
    writeln(i, '*', j, ' = ', i*j)
```
⑤ downto
```
for i := 10 downto 1 do
  writeln(i : 3)
```

①は1からnまでの和の計算であるが，このような手法は一種の定石となっているので覚えておくとよい．また②は，式1，式2が定数・変数でない例である．繰り返しが複数の文になるときは文をbegin　endで囲んで複合文とする．繰り返す文がfor文のときは多重の繰り返しとなる．④の例ではn×m回文が繰り返される．

[例題 3.5] 三平方を満たす自然数

N 以下の 3 つの自然数 A, B, C($A \leqq B \leqq C$)で, $A^2+B^2=C^2$ を満たすものを探す.

プログラム 3.5

```
{  *** 三平方を満たす自然数 ***  }
program prog3_5(input,output);
var a, b, c, n : integer;

begin
  write('最大値 ? ');
  readln(n);
  for a := 1 to n do
    for b := a to n do
      for c := b to n do
        if a*a + b*b = c*c then
          writeln('A=',a:3,'  B=',b:3,'  C=',c:3)
end.
```

◆実行例

```
最大値 ? 20
A=  3  B=  4  C=  5
A=  5  B= 12  C= 13
A=  6  B=  8  C= 10
A=  8  B= 15  C= 17
A=  9  B= 12  C= 15
A= 12  B= 16  C= 20
```

(b) 条件付き繰り返し(while, repeat 文)

繰り返しの回数が決まっているときは for 文が便利であるが, 実数型変数を制御変数として使いたい, ある条件で繰り返しを終了したい, などの場合は, while や repeat といった条件付き繰り返しが便利である.

ある条件を満たすとき, 繰り返しを行うのが while 文である. while 文の書式は次のようになっている.

while 論理式 do 文

論理式の評価結果が true の間, do の後ろの文が繰り返し実行される. 反復すべき文が複数個あ

って複合文とするときは，for 文の場合と同じように begin end で囲む．while の論理式の値が最初から false である場合には，文は一度も実行されない．また繰り返し文の実行によって論理式が true のまま変化しない場合は無限ループとなる．

例えば，変数 n を i で割り切れなくなるまで割り続けるには，次のようにする．

```
while n mod i = 0 do n := n div i
```

while 文の文例集

```
① i := n;                       n 回繰り返し
   while i > 0 do i := i-1
② i := n;
   s := 0;
   while i > 0 do begin         1〜n の和
     s := s+i;                  繰り返しが複合文
     i := i-1
   end
③ while true do write('*')     強制的無限ループ
                                止めるには CTRL-C
```

① は n 回の繰り返しで，このようにするか，次のようにする．

```
i := 0;
while i < n do i := i+1
```

② は ① の応用で 1〜n までの和を計算する例である．do の次が複合文になる．③ は強制的無限ループの例で画面に＊を書き続ける．中止するには CTRL キーを押したまま c を押す．

◆無限ループになったら...

間違えて無限ループに入ったら，CTRL-C（CTRL キーを押しながら c を押す）を押す．終了しないときは，CTRL キーを押しながら PAUSE キーを押すと終了できる．パソコンの機種によっては STOP キーを押す．また表示を一時的に止めるには CTRL-S（CTRL キーを押しながら s を押す）で止めて，他のキー例えばスペースキーを押すと表示を続ける．

repeat 文も while 文と似ているが，ある条件式が満たされない場合に繰り返しを行ない，条件式が成立した場合，繰り返しを行なわないで次の処理に進む．repeat 文の書式は，

```
repeat 文 until 論理式
```

である．

文を実行し，論理式の値が true であれば繰り返しをしないで次の処理へ進み，そうでなければ文

の実行を繰り返す.

while 文と異なり，初めに repeat と until で囲まれた文を実行し，論理式を満たしていれば次の処理に進み，それ以外の場合は文を繰り返し実行する．繰り返しの文は，repeat と until で囲まれているので，文が複数になったときでも begin end で括る必要はない.

例えば，データとして正の整数を入力すべきときに，間違えて負の数を入力しないよう正の数を入力するまで繰り返しを続けるときは次のようにする.

```
repeat
  write('正の整数?');
  readln(n)
until n > 0
```

この場合は while～do よりも repeat～until の方が使いやすい.

repeat 文の文例集

```
① i := n;            n回繰り返し
   repeat
     i := i-1
   until i = 0
② i := 1;
   s := 0;
   repeat            1～n の和
     s := s+i;       繰り返しが複合文
     i := i+1
   until i > n
③ repeat
     write('*')      強制的無限ループ
   until false       止めるには CTRL-C
```

① は n 回の繰り返しの基本形である．② は ① の応用で 1～n までの和を計算する例である．repeat until の中が複合文になる．③ は強制的無限ループの例で画面に＊を書き続ける．中止するには CTRL キーを押したまま c を押す．これら ① ② ③ は while 文の ① ② ③ と対応している.

while 文，repeat 文の練習として整数の素因数分解を行ってみる．n ≦ 1 が入力されるまで，入力・計算を繰り返す.

[例題 3.6] 素因数分解

整数 n の素因数分解を行う.

プログラム 3.6

```
{  *** 素因数分解 ***  }
program prog3_6(input,output);
var i, n : integer;
begin
  repeat
    write('n ? ');
    readln(n);
    if n <= 1 then halt;      {  終了  }
    write(n,' = ');
    i := 2;
    repeat
      while n mod i = 0 do begin
        n := n div i;
        if n = 1 then writeln(i)
                 else write(i,'*')
      end;
      i := i + 1
    until n = 1
  until false
end.
```

◆実行例

```
n ? 1234
1234 = 2*617
n ? 2345
2345 = 5*7*67
n ? 3456
3456 = 2*2*2*2*2*2*2*3*3*3
n ? 4567
4567 = 4567
n ? 5678
5678 = 2*17*167
n ? 0
```

この例題では，整数 n を整数 i で割りきれるまで割る繰り返しが while　do で表されている．また i を 2 から増やしていく繰り返しが repeat until で表されている．さらに入力と計算・表示を繰り返す無限ループの繰り返しがあり，3 重の繰り返しとなっている．終了は入力した n の値が 1 以下のとき，halt という終了手続きで終了する（無限ループから抜け出す）ようになっている．

(c)　goto 文

goto 文は繰り返しを行う文ではないが，if 文などと組み合わせて使うことにより for, while, re-

peat と同じように繰り返しを行うことができる．

goto 文があると次の文を実行しないで，goto 文に書かれたラベルにジャンプして処理の流れを変える．ジャンプ先の文には，ラベル：を付ける．ラベルは正の整数値または一般の名前(変数名と同じ)で，変数宣言と同じように label 定義しておく必要がある．別のブロック(他の手続き)にはジャンプできない．

```
program sqrt3(output);
label 55;
var x, y : real;
begin
  x:=3;
55:
  y:=x;
  x:=(x+3/x)/2;
  writeln('sqrt(3) = ',x);
  if abs(x-y)>1.e-8 then goto 55;
end.
```

このプログラムでは，処理の流れは repeat until にほぼ等しい．同じように for 文や while 文の代わりに goto 文が使えるが，goto 文を用いるとプログラム全体の構造がわかりにくくなるので，やたらに使わない方がよい．

TURBO Pascal では，プログラムの実行を停止する halt 手続きを使うことも一種の goto 文の使用と同じ効果を持つ．

◆乱数について

TURBO Pascal では乱数が使える．randomize という手続き文で初期化し，random という関数で 0 以上 1 未満の一様乱数を発生する．また random(n) のように引数を付けると，0 以上 n 未満の整数値をランダムに発生する．randomize を使わないと常に同じ乱数列が得られる．

```
randomize;
i:=random(100);    { 0以上99以下の整数 }
x:=random;         { 0以上1未満の実数 }
```

(d) 練習問題

[3-1] 0 度から 90 度までの 5 度きざみの角度に対し，sin と cos の値を表にしなさい．

[3-2] n 個のチームが総当たりで対戦するとき，対戦試合の組合せを表示しなさい．例えば 4 チームのときは，1 と 2，1 と 3，1 と 4，2 と 3，2 と 4，3 と 4 になる．

[3-3] i と j を入力し，ユークリッドの互除法で最大公約数を計算しなさい($i \geqq j$ とする)．i を j で割った余り k が 0 のとき j が最大公約数，そうでなければ i に j，j に k の値を代入して $k=0$ になるまで繰り返す．

[3-4] 1～1000 までの乱数値を 1 つ作りその数を当てなさい．x を入力して n より小さいか大きいかを表示し

て，x＝n になるまで繰り返し入力する．

ヒント：n：＝random(1000)＋1；

[3-5]　自分自身を除く約数の和がその数に等しい数を完全数と言う(例 6 ＝ 1+2+3)．2～500 までの完全数を捜してみなさい．500 を越える次の完全数は 8128 である．

[3-6]　整数 n を入力し，n 人の中の誰か 2 人の誕生日が一致する確率を計算しなさい．またその確率が 0.5 を越える最小の n を求めなさい．

[3-7]　n を入力し，次の数列の 1～n 番目の項を示しなさい．

(1)　1,2,3,4,5,6,7,8,9,10, ...

(2)　1,3,5,7,9,11,13,15,17, ...

(3)　1,2,3,1,2,3,1,2,3, ...

(4)　1,2,4,8,16,32,64, ...

(5)　1,3,6,10,15,21,28, ...(3 角数)

(6)　1,6,15,28,45,66,91, ...(6 角数)

[3-8]　次の級数和は，$\pi^2/8$ になるのでこれを利用して π の値を計算しなさい．

$s = 1+1/3^2+1/5^2+1/7^2+1/9^2\cdots$

収束は遅いので注意．

[3-9]　指数関数のテーラー展開を用いて自然対数の底の値を計算しなさい．

$\exp = 1+1/1!+1/2!+1/3!+1/4!+\cdots\cdots\cdots$

$= 2.7182818284\cdots\cdots\cdots$

[3-10]　A0 の紙の大きさは縦 840 mm×横 1188 mm であり，A1 はその半分(面積)の大きさで縦 594 mm×横 840 mm，A2 は 420 mm×594 mm である．n($0 \leqq n \leqq 8$)を入力して An の大きさを示しなさい．

3.4　配　　列

(a)　配　　列

いままで扱った変数では，1 つの変数はある瞬間には 1 つの数値しか記憶できない．しかし実際には，たくさんの数値を入力して平均や標準偏差を計算するとか，小さい順に並べ直すとか，多数のまとまった数値を効率よく扱いたいことが頻繁に起こる．このようなデータのまとまりを効率よく扱えるようにしたのが配列である．

配列の宣言は，(integer や real など型の代わりに)次のように書く．

```
array[初期値 .. 終値] of 型；
```

例：

```
var data ： array[1..100] of integer；
```

上は，data という配列名で整数型のデータを 100 個準備するという意味である．data〔1〕，data〔2〕，…，data〔100〕 までの個々の要素を変数と同じように扱うことができる．配列は必ずしも宣言した要素の個数全てを使用する必要はないので，余分に宣言してかまわない．ただし使用できる

メモリに制限があるので，この記憶量を越えることはできない．また実数型は整数型よりも多くの記憶量を必要とする．

個々の要素を指定するには，たとえばi番目の要素は，次のように表す．

変数名[i]　　または　　変数名(.i.)

iのことを添え字とも言う．添え字は整数型の式だが，その値は，もちろん初期値≦添え字の値≦終値である必要がある．したがって

data[1]　　data[i]　　data[k+3]

などのようにして，個々の要素を指定することができる．また宣言時で　初期値≦終値　の必要があるが(ただし初期値＝終値はあまり意味がない)，その値は正である必要はない．そのため，

```
const n = 3*4;
var dat1 : array[0..n] of integer;
    dat2 : array[-2..2] of real;
```

という例も考えられる．

配列を用いると，いろいろな処理のアルゴリズムを効率よく組むことができる．

```
s := 0;
for i := 1 to n do s := s+data[i]
```

というように1からn番目までのデータの和が簡単に求められる．またn次元ベクトルの内積は，次のようになる．

```
s := 0;
for i := 1 to n do s := s+v1[i]*v2[i]
```

[例題3.7]　ソーティング

数値列を入力して小さい順に並べ変える(ソーティングという)．数値列の終わりは0で示す．

プログラム3.7

```
{  *** ソーティング ***  }
program prog3_7(input,output);
var i, j, n, tmp : integer;
    dat : array[1..100] of integer;
begin
  n := 0;               {  データ入力  }
  repeat
    n := n + 1;
    write(n,' 番目の数値 ？');
```

```
10     readln(dat[n])
11   until dat[n] = 0;
12   n := n - 1;
13   for i := 1 to n - 1 do          { ソーティング }
14     for j := i + 1 to n do
15       if dat[j] < dat[i] then begin
16         tmp := dat[i];            {   データ交換   }
17         dat[i] := dat[j];
18         dat[j] := tmp
19       end;
20   write('結果=');
21   for i := 1 to n do write(dat[i]:5);
22   writeln
23 end.
```

```
◆実行例
1番目の数値 ? 23
2番目の数値 ? 8
3番目の数値 ? 4
4番目の数値 ? 15
5番目の数値 ? -1
6番目の数値 ? 0
結果=    -1    4    8   15   23
```

この実行例の場合，入力直後の配列の内容は次のようになっている．

i=1　　j=2

dat[1]	dat[2]	dat[3]	dat[4]	dat[5]
23	8	4	15	-1

n=5

繰り返し処理で i = 1, j = 2 のとき dat[1] > dat[2] であるから，両者を交換する．2つの値を交換するには，プログラムの 16〜18 行のように別の変数* に一方の値を保存しておかなければならない(tmp は tempolary の意味)．

i=1　　　　j=3

dat[1]	dat[2]	dat[3]	dat[4]	dat[5]
8	23	4	15	-1

同じように，i = 1, j = 3 で dat[1] > dat[3] であるから，dat[1] と dat[3] の値を交換する．i = 1, j = 4 では dat[1] < dat[4] であるから値の交換はない．i = 1, j = 5 では dat[1] と dat[5] の値を交換する．i = 1 の繰り返しが終わり dat[1] には最小値が入っている．

* 両者の差分を計算すれば別の変数を使わずに済むが，exor か加減算が必要になる．

i=1　　　　　　　　　　　　　j=5

dat[1]	dat[2]	dat[3]	dat[4]	dat[5]
-1	23	8	15	4

同じように i = 2 に設定して j = 3 から j = 5 までの繰り返しが終わり dat[2] には 2 番目に小さい値が入る．繰り返しがすべて終了すると，配列の数値が小さい順に並ぶことがわかる．

dat[1]	dat[2]	dat[3]	dat[4]	dat[5]
-1	4	8	15	23

◆添字の値

var x : array[1..10] of real;　と定義した場合，添字の範囲は 1～10 に限られる．しかしプログラムのミスなどから，i = 0 とか i = 11 のときに x[i] を参照したり，x[i] に値を代入することがある．このときエラーが検出されないことがあるので，これをチェックするためには，プログラムの先頭に {$R+} というコンパイラオプションをいれておく．ただし実行は少し遅くなる．

(b)　2 次元配列

上のような数値列のことを 1 次元配列という．この他行列のような $A_{i,j}$ の形(2 次元配列)のデータやより多次元の $B_{i,j,k}$ の形(3 次元配列)のデータを利用することもできる．

```
var matrix : array[1..m] of array[1..n] of real;
```

n 個の配列が m 個あるときこのように書けるが，

```
var matrix : array[1..m, 1..n] of real;
```

というように書くこともできる．例えば，

```
var a, b, c : array[-1..4, 0..2, 1..8] of real;
```

これは，6×3×8 = 144 個の実数型データを持つ 3 次元の配列 a, b, c を宣言したことになる．3 次元配列の各要素も 2 次元配列とおなじように表すことができる．

```
a[1, 1, 1]    b[i, j, k]    c[i+2, 3*j, k div 2]
```

例としてベクトル・行列の積を考えると，次のようになる．

```
for i := 1 to n do begin
  x[i] := 0;
  for j := 1 to n do
    x[i] := x[i] + a[i, j] * b[j]
end
```

また行列どうしの積は次のようになる．

for i：＝1 to n do

for j：＝1 to n do begin

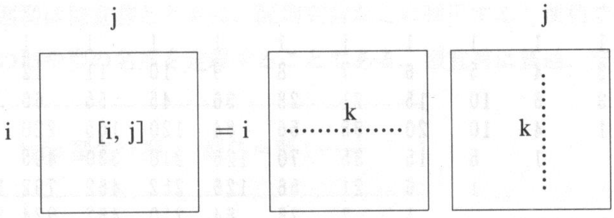
x[i, j]：＝0；

for k：＝1 to n do

x[i, j]：＝x[i, j]＋a[i, k]＊b[k, j]

end

end



[例題 3.8]　パスカルの三角形

パスカルの三角形を 2 次元配列で表し，転置して表示する．

1	これを	p[1,1]
1 1		p[2,1]p[2,2]
1 2 1		p[3,1]p[3,2]p[3,3]
1 3 3 1		p[4,1]p[4,2]p[4,3]p[4,4]
1 4 6 4 1		p[5,1]p[5,2]p[5,3]p[5,4]p[5,5]
…………………		…………………………

左のパスカルの三角形を右のように 2 次元配列でこのように表す．計算は足し算だけで済む．表示の際，見やすいように転置して表示する．

プログラム 3.8

```
{  *** パスカルの三角形 ***  }
program prog3_8(input,output);
var i, j, n : integer;
    p : array[1..20,1..20] of integer;

begin
  write('n ? ');
  readln(n);
  for i := 1 to n do begin      {  計算  }
    p[i,1] := 1;
    p[i,i] := 1;
    for j := 2 to i-1 do
```

```
      p[i,j] := p[i-1,j-1]+p[i-1,j]
  end;
  for j := 1 to n do begin          { 表示 }
    for i := 1 to n do
      if i>=j then write(p[i,j]:5)
        else write(' ':5);
    writeln
  end
end.
```

◆実行例(n=14)

```
    1    1    1    1    1    1    1    1    1    1    1    1    1    1
         1    2    3    4    5    6    7    8    9   10   11   12   13
              1    3    6   10   15   21   28   36   45   55   66   78
                   1    4   10   20   35   56   84  120  165  220  286
                        1    5   15   35   70  126  210  330  495  715
                             1    6   21   56  126  252  462  792 1287
                                  1    7   28   84  210  462  924 1716
                                       1    8   36  120  330  792 1716
                                            1    9   45  165  495 1287
                                                 1   10   55  220  715
                                                      1   11   66  286
                                                           1   12   78
                                                                1   13
                                                                     1
```

◆配列の初期値設定について

TURBO Pascal では配列の初期値設定を行うことができる．型付き定数定義の初期値設定はコンパイルのとき行われるので，プログラムで値が変更されると，2 回目以降初期値が変わってしまうので，注意すること．

例：

```
const a : array[1..3] of integer = (1,2,3);
      b : array[1..2,1..5] of integer = ((1,2,3,4,5),(2,3,4,5,6));
```

(c) 順 序 型

case 文や配列の宣言に使われる'数値..数値'は部分範囲型という一種の整数型のデータ型を定義する．整数型にはデータの値の範囲から integer, longint, word, byte, shortint の 5 種類がある (TURBO Pascal ver. 5 の場合)．これ以外に部分範囲型と列挙型という 2 種類の特別な型がある．

型名のところが

値 1.. 値 2

であれば，値1以上値2以下の整数値に限られる部分範囲型になる．実際のデータ型は，byte, integer, longintなどになる．値1～値2の範囲を満たす最小のデータ型になる．例えば0～255に収まっていればbyte型，－10000～10000であればinteger型になる．演算や入出力に関しては一般の整数型と同じである．

例：

```
var a, b, c : 1..1000 ;
    x, y, z : -1..127 ;
```

部分範囲型は型宣言とともに，配列宣言などに利用すると便利である．型宣言というのは，自分でデータの型や型の名前を定義することである．型宣言は普通，定数宣言と変数宣言の間で行う．

type 型名＝型；　型名＝型；……

たとえば，次のような型宣言があれば，変数宣言で型定義された名前を用いることができる．

```
type seisu = integer ;   jissu = real ;
     takusan = array[1..100] of integer ;
var  i, j, k : seisu ;          整数型
     a, b, c : jissu ;          実数型
     x, y, z : takusan ;        整数型配列
```

したがって配列の宣言に使うには次のように用いる．

```
const size = 10 ;
type dim = 1..size ;
var   vector : array[dim] of real ;
      matrix : array[dim, dim] of real ;
```

部分範囲型とは別に，列挙型と呼ばれるデータの型がある．名前のついた少ない数のまとまりを表すのが列挙型である．列挙型はスカラ型とも呼ばれる．列挙型の定義は次のように異なる名前をコンマで区切って全体を(　)で囲む．

(名前，名前，……，名前)

例：

```
type suuji = (zero, one, two, three) ;
var sc1, sc2 : suuji ;
```

列挙型変数に直接数値を代入することはできない．入出力や演算を行うこともできない．宣言された項目の名前を代入すること，同じ名前の列挙型変数どうしで値を転送することはできる．ただし型変換関数を使うと列挙型と整数型などと数値を交換することができる．

i：＝ord(zero)；

sc1：＝suuji(i)；

上の例では宣言された項目は先頭から0，1，2……の整数値になる．下はその逆であり，宣言した列挙型の型名が整数から列挙型への変換を行う関数名になる．

整数型，部分範囲型，列挙型，文字型(3.7参照)のことをまとめて順序型という．順序型はたとえばfor文の制御変数として用いることができる．

for ch：＝'a' to 'z' do…… 文字型

for sc1：＝one to three do…… 列挙型

(d) 練習問題

[4-1] 整数n(n > 0)をk進法(1 < k < 10)で表しなさい．

[4-2] 数値列(終わりは0)を入力し，平均値と平均値以上の個数，平均値未満の個数を表示しなさい．

[4-3] 月別降水量を配列に入れて棒グラフを表示しなさい．棒は縦棒とし，棒の長さは大まかでよい．

月	1	2	3	4	5	6	7	8	9	10	11	12
mm	54	63	102	128	148	181	125	137	193	181	93	56

[4-4] a月b日の1週間(7日)後の月日を求めなさい(閏年ではないとする)．

ヒント：1～12月の日数を配列に入れておく．

[4-5] ある催し物がa月b日からc月d日まで行われるときの開催期間(日数)を求めなさい(a月とc月は同じ年で閏年ではないとする)．

[4-6] 例えば128をx[3]＝1, x[2]＝2, x[1]＝8と表す．このようにして2の100乗を計算しなさい．

[4-7] 200個の整数型配列に1～200の数値を代入し，それらの数値が縦に並ぶ形式の表を作りなさい．はじめにnを入力し，横にn個並ぶようにしなさい．

[4-8] 整数1234の回文数は4321である．回文数も同じ桁数で，その数が回文数で割り切れる数をいくつか捜しなさい．

[4-9] 整数n(longint型)を入力して金額を表示するときのように，3桁ごとにコンマを打って表示しなさい．

例：12345678 → 12, 345, 678

[4-10] xのn次多項式 $f(x)=a_n x^n+a_{n-1}x^{n-1}+\cdots+a_1 x+a_0$ がある．各係数 a_i の値は配列a[i] に代入されていて，f＝a[n]，g＝0として，n＝n−1，g＝g∗x+f，f＝f∗x+a[n] を繰り返して，関数値と微分値を計算するのがホーナー法である．$f(x)=x^3-x$ の $-1 \leqq x \leqq 1$ の区間(xの間隔は0.1とする)について，x, f(x)，f'(x)の値を表にしなさい．

3.5 関　　数

(a) 関　　数

いままでにtrunc，roundなどの変換関数やsin，sqrtなどの算術関数を扱った．関数の使い方は，

y：＝sin(x)

などと記述することにより，関数名の(　)の中に関数に渡す数値(式)を書き，その値に対する関数値が計算されて変数yに代入される．

このような関数のプログラムを自分で組むことができる．次の例では，x の n 乗を計算する関数を作り，それを円の面積や球の体積計算に利用している．

```
 1 program en_kyuu(input,output);
 2 var r,s:real;
 3
 4 function beki(x,n:real):real;    {  関数  }
 5 begin
 6   beki:=exp(n*ln(x))
 7 end;
 8
 9 begin                  {  メインプログラム  }
10   write('半径 ? '); readln(r);
11   s:=pi*beki(r,2);
12   writeln('円の面積 = ',s);
13   writeln('球の表面積 = ',4*pi*beki(r,2));
14   writeln('球の体積 = ',4/3*pi*beki(r,3))
15 end.
```

このように，function…から次の複合文 begin～end；までが関数 beki の定義部分である．実行は関数の定義部分を飛び越えて begin の次の文から始まる．関数の名前があると，その関数が呼び出される．

関数部分のプログラムは，

```
function 名前(引数宣言)：結果型；
宣言部
begin　実行部分　end；
```

という形をしている．関数の名前は変数などと同じ規則で決める(1.3 参照)．関数値が何型かを示す「：結果型」が後ろにつく．結果型は必要に応じて integer，real，boolean などになる．宣言部では，関数値の計算に必要な定数や変数の宣言を行う．実行部分には関数の値を返すため，「関数名：＝式」などの文を含む．

関数には，引数のない場合とある場合がある．関数頭部の例をいくつか示す．

```
function fc1：real；                    引数がない例
function fc2(x, y：real)：integer；
function fc3(x, y, z：real；
             i, j：integer; b：boolean)：boolean；
```

プログラム後半の(上のプログラムでは 9～15 行)begin　end. で囲まれた実行部分をメインプログラムと言う．関数部分も 1 つのプログラムなので，副プログラムと言う．プログラムが実行され

る場合は，メインプログラムから実行がはじまり，関数の参照があると，その関数の実行部分が実行される．関数の実行部分が終了すると，再びメインプログラムに処理が戻る．

[例題 3.9] 二 分 法

二分法で $f(x)=x^2-\cos(x)$ の解を求める．$f(0)<0$，$f(\pi/2)>0$ であるからその間に $f(x)=0$ となる x が存在する．

プログラム 3.9

```
{  *** 二分法 ***  }
program ex3_9(output);
const eps = 1.e-6; max = 100;
var i : integer;
    a, b, x : real;

function f(x:real):real;    {  関数f(x)  }
begin
    f := x*x - cos(x)
end;

begin
  i := 0;
  a := 0.0;
  b := pi/2;
  writeln('i':3,'a':7,'b':10,'f(x)':12);
  repeat
    x := (a + b)/2;
    writeln(i:3,a:10:5,b:10:5,f(x):10:5);
    if f(x)<0 then a:=x else b:=x;
    i := i + 1
  until (i>=max) or (abs(f(x))<eps);
  writeln('解= ',x)
end.
```

◆実行例

i	a	b	f(x)
0	0.00000	1.57080	-0.09026
1	0.78540	1.57080	1.00523
2	0.78540	1.17810	0.40826
3	0.78540	0.98175	0.14631
4	0.78540	0.88357	0.02481
5	0.78540	0.83449	-0.03353
6	0.80994	0.83449	-0.00457
⋯⋯			
15	0.82413	0.82418	0.00005

```
 16   0.82413   0.82416   0.00003
 17   0.82413   0.82414   0.00001
 18   0.82413   0.82414   0.00000
 19   0.82413   0.82413   0.00000
解=  8.2413267632E-01
```

関数f(x)が連続であればf(a)＜0, f(b)＞0に対しf(x)＝0を満たすxがaとbの間に一つ以上存在する．xを効率よく捜すためa, bの区間を半分ずつ狭めていくのが二分法である．x＝(a+b)/2として，f(x)が正であればaとxの間，f(x)が負であればxとbの間に解が存在する．|f(x)|が十分小さくなるまで繰り返す．

f'(x)が計算でき，f'(x)≠0のときはNewton法が使える(第4章参照)．二分法より，Newton法の方が収束ははるかに速い．

(b) プログラムの構造

関数functionの記述はprogram頭部とメインプログラムの間に置く．関数が複数個ある場合は並列に並べる．関数が別の関数を呼び出す場合，Turbo Pascalでは呼び出される関数は呼び出す関数の前に位置していなければならない．副プログラムがあるプログラムで変数を使用する場合，変数の宣言場所に注意しなければならない．

下の図で初めに宣言された変数A, B, Cは，関数でもメインプログラムでも参照できる共通の変数になる．関数内だけで使用する変数D, E, Fは，その関数の頭部と実行部分との間で宣言する．関数の後で宣言された変数J, K, Lは，メインプログラム内でのみ使用できる．したがって変数A, B, Cはすべての副プログラムで共通に使用できる共通変数である(グローバル変数)．変数D, E, Fなどは局所(ローカル)変数と呼ばれる．

```
program pro1････;
var A,B,C･･･                共通変数

function func1(X,Y,Z････):型;
var D,E,F･･･                関数内変数
begin
  ･･･････                 変数  A,B,C  D,E,Fが参照できる
end;

var J,K,L･･･                メインプログラム内
begin
  ････････                変数  A,B,C  J,K,Lが参照できる
  ･･func1(S,T,U･･)･･･
end.
```

引数 X, Y, Z は A, B, C と同じ名前であってもよいが，その場合，共通変数は参照できなくなる．同じように変数名 D, E, F と A, B, C は同じ名前でもよいが，共通変数は参照できなくなる．ただし同じ関数内では，引数 X, Y, Z と局所変数 D, E, F は同じ名前であってはならない．

メインプログラム内の関数呼び出しでは，引数 X, Y, Z に対して S, T, U を設定する．X, Y, Z を仮引数，S, T, U を実引数という．X, Y, Z のデータの型や引数の個数に対し，S, T, U の方も型，個数を一致させなければならない．ただし，仮引数が実数型の場合には実引数は整数型でもよい．

実引数 S, T, U は，式が許されるので変数でも，定数でも計算式でもよい．変数の場合は A, B, C または J, K, L のどれかである．後で説明するように仮引数が変数引数の場合は，実引数は変数に限られる．

◆forward 宣言

TURBO Pascal では呼び出される関数は呼び出す関数の前に位置していなければならないが，最初に関数の頭部だけを forward 宣言しておくと，順序不動でよい．

```
function fc1 (x : integer ; n : real) : real ; forward ;
```

(c)　再帰呼び出し

関数内部で自分自身を呼び出すのが再帰呼び出しである．再帰呼び出しを利用すると，漸化式のプログラムなどを簡単に組むことができる．たとえば 1 から n までの和，階乗やフィボナッチ数列は，

$$\Sigma n = \Sigma (n-1) + n, \quad \Sigma 1 = 1$$

$$n ! = (n-1) ! \cdot n, \quad 1 ! = 1$$

$$F(n) = F(n-1) + F(n-2), \quad F(0) = 0, F(1) = 1$$

などの漸化式で表わすことができる．これらの漸化式は，ほとんどこのままプログラムとして記述することができる．例えば，再帰呼び出しを用いて 1 から n までの和を求める関数は次のようになる．

```
function sum (n : integer) : integer ;
begin
  if n = 1 then sum := 1
           else sum := sum (n-1) + n
end ;
```

これらの例は再帰にしても計算時間などの点で繰り返しより特に利点はない．次の例は x の n (n > 0 の整数) 乗を計算する関数である．再帰呼び出しを使い，n が奇数，偶数で場合分けをする (偶数の場合は 2 乗する)．

[例題 3.10]　x の n 乗

実数 x の n 乗(n > 0 の整数型)を再帰的関数で求める．

プログラム 3.10

```
1  { *** xのn乗 *** }
2  program prog3_10(input,output);
3
4  function f(x:real;n:integer):real;
5  begin
6    { writeln(n); }
7    if n=1 then f:=x else
8      if n mod 2=0 then f:=sqr(f(x,n div 2))
9                   else f:=f(x,n-1)*x
10 end;
11
12 var x:real; n:integer;
13 begin
14   write('x n ? ');
15   readln(x,n);
16   writeln(f(x,n):10:8)
17 end.
```

6 行目のコメントを外すと({と} を消す)途中の呼び出し経過がわかる．たとえば n = 7 のとき，次のように展開して計算すると考えればよい．

$$
\begin{aligned}
f(x, 7) &= f(x, 6) * x \\
&= sqr(f(x, 3)) * x \\
&= sqr(f(x, 2) * x) * x \\
&= sqr(sqr(f(x, 1)) * x) * x \\
&= sqr(sqr(x) * x)) * x
\end{aligned}
$$

この例では単純な繰り返し計算に比べて乗算回数はほとんど変化ないが，たとえば n = 100 では乗算回数は 1/10 に減少する．

(d)　練習問題

[5-1]　常用対数を計算する関数 log 10(x) を作り，整数の桁数を求めなさい．

[5-2]　2 つの引数 xy の大きい方の値を返す関数 f を作り，メインプログラムで a, b, c, d の 4 つの値を入力して　f(f(…)) という形でその最大値を求めなさい．

[5-3]　再帰呼び出しで配列の 1～n までの要素の和を求めなさい．

[5-4]　二分法または Newton-Raphson 法(第 4 章参照)を用いて立方根を求める関数を作りなさい．

[5-5]　ユークリッドの互除法で最大公約数を求める関数を作りなさい(練習問題 3-3 参照)．再帰呼び出しを使うことにする．

[5-6]　再帰的関数でフィボナッチ数列を求めなさい．

[5-7] 角度(度)をラジアンに変換する関数 radian，および tangent を求める関数 tan を作り，これらを利用して tan x° を計算しなさい．

[5-8] 配列の n 個のデータの最大値を求める関数 maxdat を作りなさい．配列は共通変数とし，maxdat(n) は maxdat(n−1) と n 番目のデータを比較する再帰的関数とする．

[5-9] 例題 3.10 の関数を利用して $n = 10^m$，$1 \leq m \leq 5$ のときの $(1+1/n)^n$ の値を求めなさい．

[5-10] 引数 y を西暦年として閏年なら true，閏でないなら false を返す関数 leap を作りなさい(練習問題 2-8 参照)．

3.6 手続き

(a) 手続き

関数 function の仲間に手続き procedure がある．関数も手続きもある処理を定義する副プログラムである．関数と手続きの違いは，関数は値を返すので式の中で呼び出すのに対し，手続きは手続き文で呼び出す．

```
program proctest(output);

procedure bar(x:integer);    { 手続き }
var i:integer;
begin                        { 棒グラフ }
  for i:=1 to x do write('*');
  writeln
end;

begin                        { メインプログラム }
  bar(10);                   { データの表示 }
  bar(20);
  bar(30)
end.
```

手続きには，関数と同じように名前をつける．引数も関数の場合と同じだが，関数値を返さないので関数値の型を宣言する必要はない．したがって次のような構造となる．

```
procedure 名前 (引数宣言);
宣言部分
begin  実行部分  end;
```

手続きと関数の違いは，関数では関数値を返すために，関数の型を定義したり，関数名に値を代入したりするが，手続きでは不要である．また呼び出す場合，関数は式の中で関数を呼び出すが，

手続きは手続き文で呼び出す.

プログラムが実行される場合は，メインプログラムから実行がはじまり，手続き文があると，その手続きの実行部分が実行される．手続きの実行部分が終了すると，再びメインプログラムの手続き文の次の文に処理が移る.

データの受け渡しがあるときは関数同様引数を利用してこれを行なうことができる．その場合手続き名の後ろに次のような引数宣言部を付ける.

procedure proc1(a, b, c : real);

procedure proc2(x, y : real ; i, j : integer)

引数は呼び出す側では，実引数と呼び，呼ばれる側では仮引数と呼ぶ．引数をもつ手続きを呼び出す場合は，関数と同じように手続き文に実引数を付けて呼び出す.

proc1(a, 1.5, 2*x);

proc2(a, b, i, j);

また関数と同じように自分自身を呼び出す再帰的手続きを定義することもできる.

[例題 3.11] 順列の生成

1〜n の順列をすべて表示する．n 個の数値を巡回させると n 通りの順列になるので，それぞれについて n−1 の順列を考えると n! の全順列が求められる.

プログラム 3.11

```
{  *** 順列の生成 ***  }
program prog3_11(output);
var p: array[1..100] of integer;
    n, i : integer;

procedure perm(m:integer);
var i, j, k : integer;
begin
  if m>1 then
    for i := 1 to m do begin
      perm(m-1);            {  m-1 の順列 }
      k := p[1];            {  ローテート  }
      for j := 1 to m-1 do p[j] := p[j+1];
      p[m] := k
    end
  else begin            {  表示  }
    for i := 1 to n do write(p[i]:3);
    writeln
  end
end;
```

```

begin                    { メインプログラム }
  n := 4;
  for i := 1 to n do p[i] := i;
  perm(n)
end.
```

◆実行結果

```
1 2 3 4          1 4 3 2
2 1 3 4          4 1 3 2
1 3 2 4          1 3 4 2
3 1 2 4          3 1 4 2
3 2 1 4          3 4 1 2
2 3 1 4          4 3 1 2
1 2 4 3          4 2 3 1
2 1 4 3          2 4 3 1
1 4 2 3          4 3 2 1
4 1 2 3          3 4 2 1
4 2 1 3          3 2 4 1
2 4 1 3          2 3 4 1
```

このプログラムでは，手続きを再帰的に呼び出している．4つの数のローテートを考えると，4通りの状態があり，4回目の操作で元に戻る．

1 2 3 4 (1)

2 3 4 1 (2)

3 4 1 2 (3)

4 1 2 3 (4)

1 2 3 4 元に戻る

これらそれぞれについて最初の3個の順列を考える．(1)については，3通りある．

1 2 3 (1.1)

2 3 1 (1.2)

3 1 2 (1.3)

1 2 3 元に戻る

最初の2個の順列を考えると，2通りある．

1 2 (1.1.1)

2 1 (1.1.2)

1 2 元に戻る

このように全順列を考えると $n\cdot(n-1)\cdot(n-2)\cdots\cdots=n!$ の組合せがある．

(b) 引　数

標準に用意されている手続きの代表的なものに read と write がある．この2つの大きな違いは引数が変数引数と値引数というところである．

```
read(a, b, c);            変数引数
write(a, 2*b, c/3);       値引数
```

つまり，read 文の場合は引数は変数に限られ式は使えない．write 文の場合は値を渡すので引数に定数や式が使えた．また read 文では呼び出しによって実引数の変数に値が代入される．

このように仮引数は大きく分けると2種類ある．今まで扱ってきた引数は値引数で，これを変数引数にすることもできる．変数引数にするには，宣言時に前に var を付ける．

```
function func1(var n : integer) : boolean;
procedure proc1(a, b, c : real; var x, y, z : real;
                i, j : integer; var m, n : integer);
```

これを呼び出す場合も変数の型，変数引数・値引数の違いに注意しなければならない．上の proc1 の場合，はじめの3つが実数型の値引数，次の3つが実数型の変数引数，……という引数の型，個数，順番などに注意する．

```
proc1(a, 1.5, 2*c, x, y, z, 23, n mod 2, i, j)
```

手続き proc1 の中では，変数引数 x, y, z, m, n に値を代入するとそれが実引数の値となる．例えば，手続き内で引数 n に5を代入すると，呼び出した後に変数 j に5が代入される．

このように，値引数と変数引数の見かけ上の違いは仮引数の先頭に var があるかないかの違いである．値引数は手続き側で引数の値を参照するだけであり，その結果を呼び出した側に戻す場合は，変数引数としての宣言を行わなければならない．値引数を使用する場合は，実引数に式や定数を割り当てることができる．

	値引数	変数引数
呼び出す方	式(変数，定数)	変数のみ
手続き内	値を返せない	値を返せる

実引数/仮引数の対応では引数の個数と型を揃えることに注意する．また引数が配列の場合は，できるだけ変数引数を用いる．TURBO Pascal で引数に配列を使う場合は，前もって配列の型宣言をしておかなければならない．

```
program test(input, output);
type data = array[1..100] of integer;
  ..................
procedure p1(var buf : data);        ←ここに array などの宣言は使えない
begin
  ..................
```

次に配列の引数を用いてデータの受け渡しを行う例を示す．

[例題 3.12] エラトステネスの篩

自然数2～nに1を付けておき，2の倍数，3の倍数，……に0を付けていく(自分を除いて)．残った数は素数の集合になる．

プログラム 3.12

```
{ *** 素数表 (エラトステネスの篩) *** }
program prog3_12(output);
const n=1000;
type data=array[2..n] of integer;

procedure disp(var prime:data);
var i:integer;
begin
  for i:=2 to n do
    if prime[i]<>0 then write(i:5);
  writeln
end;

procedure mark(i:integer;var prime:data);
var k:integer;
begin
  k:=2*i;
  while k<=n do begin
    prime[k]:=0;
    k:=k+i
  end
end;

var prime:data; i:integer;
begin
  for i:=2 to n do prime[i]:=1;
  for i:=2 to n do
    if prime[i]<>0 then mark(i,prime);
  disp(prime)
end.
```

◆実行例（部分）

```
    2    3    5    7   11   13   17   19   23   29   31   37   41   43
   47   53   59   61   67   71   73   79   83   89   97  101  103  107
  109  113  127  131  137  139  149  151  157  163  167  173  179  181
  191  193  197  199  211  223  227  229  233  239  241  251  257  263
  269  271  277  281  283  293  307  311  313  317  331  337  347  349
  353  359  367  373  379  383  389  397  401  409  419  421  431  433
```

このプログラムでは，はじめに配列の全要素に1を代入しておく(次の表参照)．次にi＝2について2の倍数4，6，8，10……に0を代入する．次に3について，3の倍数6，9，12，15，18，……に0を代入する．4はすでに0になっているので，次の5について5の倍数10，15，20，25，……に0を代入する．これをnまで繰り返せば，配列の要素番号が素数のとき1，それ以外のとき0が代入されている．これを表示すれば素数表ができあがる．この方法は数字1つ1つが素数であるかどうかを判定する方法に比べ高速に素数表を作成することができる．

添　字	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
初期値	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=2	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0
i=3	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0
i=5	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0
i=7	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0

ここでは2以上の数を調べたが，普通は3以上の奇数を調べる．この例題のようにTURBO Pascalでは，引数に配列を使用する場合は，必ず型定義をおこなわなければならない．

(c)　プログラムの構造

ここで，いままででてきた宣言の種類をまとめると次のようになる．手続きや関数も一種の宣言である．

label	ラベル定義
const	定数定義
type	型定義
var	変数宣言
procedure	手続き宣言
function	関数宣言

このように手続きや関数の宣言部に，別の手続きや関数が定義されることも考えられる．したがって，手続きや関数は次のように’入れ子’になることもある．この場合，変数の有効範囲や，入れ子になった手続き・関数の呼出に注意する．

```
program pro1;
var A,B,C･･･

procedure proX･･･
var D,E,F･･･
  function funcY･･･
  var G,H,I･･･
  begin
    実行部分(A1)        変数 A,B,C D,E,F G,H,I が参照可
  end;
begin
  実行部分(A)          変数 A,B,C D,E,F が参照可
end;                   関数 A1 が呼出可

var L,M,N･･･              メインプログラム
begin
  実行部分             変数 A,B,C L,M,N が参照できる。
end.                   Aは呼出可、A1は不可
```

(d) 練習問題

[6-1] 以前扱ったソーティングのプログラム(例題3.1)を，手続きを使うように(手続き内でソートする)書き換えなさい．データ(配列)を共通変数とする．

[6-2] [6-1]と同じ問題．データ(配列)を引数で手続きに渡しなさい．

[6-3] 2つの分数a/bとc/dの四則演算を行いなさい．a, b, c, dは整数型とし，結果が約分できる場合は約分する．

[6-4] 例題3.12の配列を整数型から論理型に変えなさい．また3と5，5と7のように差が2の素数の組(これを双子素数という)をすべて表示しなさい．

[6-5] 配列p[1]，…，p[n] があり，引数j, k($1 \leqq j \leqq k \leqq n$)に対しp[j]，p[j+1]，…，p[k−1]，p[k] をp[k]，p[k−1]，…，p[j+1]，p[j] と逆順に並べ変える手続きを作りなさい．

[6-6] 前問の手続きを利用してa, b, dを与えて($1 \leqq a \leqq b \leqq d \leqq n$)，p[a]，…，p[b]，p[c]，…，p[d]，c = b+1をp[c]，…，p[d]，p[a]，…，p[b] と並べ変える手続きを作りなさい．

ヒント：a~b, c~d, a~dを順に逆順にする．

[6-7] 2つの行列の積・和を計算する手続きを作りなさい(3.4参照)．

[6-8] 前問の手続きを利用して，

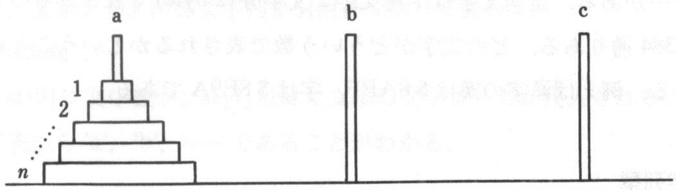
$$\exp\begin{pmatrix} 0 & -\theta p_3 & \theta p_2 \\ \theta p_3 & 0 & -\theta p_1 \\ -\theta p_2 & \theta p_1 & 0 \end{pmatrix}, \qquad p_1{}^2+p_2{}^2+p_3{}^2=1$$

が収束して(p_1, p_2, p_3)軸のまわりのθの回転を行う行列になることを確かめなさい．

[6-9] n×n行列(2次元配列)のi行をn次元ベクトル(1次元配列)として引数で返す手続きmati，j列をn次元ベクトルとして返す手続きmatj，2つのベクトルの内積を返す関数veciを作り，それを利用して2行列の積を計算しなさい．

[6-10] 再帰呼び出しを使ってハノイの塔の問題を解きなさい．ハノイの塔とはa, b, c 3つの棒があり，aの棒だけに大きい円盤から順に小さい円盤がn枚積み重ねてある．円盤は一度に1枚だけ動かせる．小さい円盤の上に大きい円盤を乗せずに全部の円盤をcの棒に移す．

ヒント：n−1枚の円盤をa→bに移し，n枚目をa→cに移すと表示し，n−1枚の円盤をb→cに移す．



3.7　文字とファイルの取り扱い

(a)　文字コード

一般の文字(半角文字)には次の表のような文字があり，それをどのような数値(コード)で表すかはあらかじめ定められている．英数字・記号の部分はASCII(アスキー)コードという．この表の文字コードは，16∗H＋Lで表される．Hは上位4bitの表わす16進数，Lは下位4bitの表わす16進数である．

例えば数字の0は，16∗3＋0＝48となる．これを16進法で表すと，H＋L＝$30となる．先頭の$は16進を表す記号．アルファベットのAは16進法で$41，10進法で16∗4＋1＝65となる．文字mは16進法で$6D，10進法で16∗6＋13＝109となる．

下位4bit（16進法）

<table>
<tr><th></th><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th></tr>
<tr><td rowspan="16">上位4bit 16進法</td><td>0</td><td colspan="16">制御文字領域</td></tr>
<tr><td>1</td><td colspan="16">制御文字領域</td></tr>
<tr><td>2</td><td></td><td>!</td><td>"</td><td>#</td><td>$</td><td>%</td><td>&</td><td>'</td><td>(</td><td>)</td><td>*</td><td>+</td><td>,</td><td>-</td><td>.</td><td>/</td></tr>
<tr><td>3</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>:</td><td>;</td><td><</td><td>=</td><td>></td><td>?</td></tr>
<tr><td>4</td><td>@</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td><td>H</td><td>I</td><td>J</td><td>K</td><td>L</td><td>M</td><td>N</td><td>O</td></tr>
<tr><td>5</td><td>P</td><td>Q</td><td>R</td><td>S</td><td>T</td><td>U</td><td>V</td><td>W</td><td>X</td><td>Y</td><td>Z</td><td>[</td><td>¥</td><td>]</td><td>^</td><td>_</td></tr>
<tr><td>6</td><td>`</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td><td>h</td><td>i</td><td>j</td><td>k</td><td>l</td><td>m</td><td>n</td><td>o</td></tr>
<tr><td>7</td><td>p</td><td>q</td><td>r</td><td>s</td><td>t</td><td>u</td><td>v</td><td>w</td><td>x</td><td>y</td><td>z</td><td>{</td><td>|</td><td>}</td><td>‾</td><td></td></tr>
<tr><td>8</td><td colspan="16">全角文字用</td></tr>
<tr><td>9</td><td colspan="16">全角文字用</td></tr>
<tr><td>A</td><td></td><td>｡</td><td>｢</td><td>｣</td><td>､</td><td>･</td><td>ｦ</td><td>ｧ</td><td>ｨ</td><td>ｩ</td><td>ｪ</td><td>ｫ</td><td>ｬ</td><td>ｭ</td><td>ｮ</td><td>ｯ</td></tr>
<tr><td>B</td><td>ｰ</td><td>ｱ</td><td>ｲ</td><td>ｳ</td><td>ｴ</td><td>ｵ</td><td>ｶ</td><td>ｷ</td><td>ｸ</td><td>ｹ</td><td>ｺ</td><td>ｻ</td><td>ｼ</td><td>ｽ</td><td>ｾ</td><td>ｿ</td></tr>
<tr><td>C</td><td>ﾀ</td><td>ﾁ</td><td>ﾂ</td><td>ﾃ</td><td>ﾄ</td><td>ﾅ</td><td>ﾆ</td><td>ﾇ</td><td>ﾈ</td><td>ﾉ</td><td>ﾊ</td><td>ﾋ</td><td>ﾌ</td><td>ﾍ</td><td>ﾎ</td><td>ﾏ</td></tr>
<tr><td>D</td><td>ﾐ</td><td>ﾑ</td><td>ﾒ</td><td>ﾓ</td><td>ﾔ</td><td>ﾕ</td><td>ﾖ</td><td>ﾗ</td><td>ﾘ</td><td>ﾙ</td><td>ﾚ</td><td>ﾛ</td><td>ﾜ</td><td>ﾝ</td><td>ﾞ</td><td>ﾟ</td></tr>
<tr><td>E</td><td colspan="16">全角文字用</td></tr>
<tr><td>F</td><td colspan="16">全角文字用</td></tr>
</table>

全体で16∗16＝256文字あるが，このうち32文字は制御文字，64文字は全角文字用となっている．全角文字は漢字，ひらがな，カタカナなどで，アルファベットも全角文字のABCD……と

半角文字の ABCD……がある．全角文字は半角文字 2 文字分 (2 byte) で表されるので，組み合せとして 64 ∗ 256 = 16,384 通りある．どの文字がどういう数で表されるかということ (コード) はあらかじめ定められている．例えば漢字の漢は＄8ABF，字は＄8E9A である．

(b) 文字・文字列型

Pascal のプログラムでは，文字や文字列を容易に扱えるようになっている．文字型と文字列型の 2 つのデータの型がある．文字型では，半角文字 1 文字，文字列型では全角文字や複数の半角文字 (255 文字以下) が扱える．型名は char, string である．

```
program moji(input,output);
var ch:char;
begin
  ch:='a';  ch:=#67;     { 代入例 }
  ch:=#$5F;         { 16進 }
  readln(ch);     { キーボード入力 }
  ch:=chr(ord('A')+1);   { 次の文字 }
  writeln(ch,'=',ord(ch))    { 表示 }
end.
```

文字データは，文字列を引用符で括って表わす．コードの数値で文字を与えるには ＃78 のように ＃ をつけて表現する．文字型から文字コードを得るには，

code：= ord (a);　　(code は整数型)

ch：= chr (code);　　(ch は文字型)

のように，ord という関数を用いる．文字コードから文字を得るには，chr という関数を用いる．

文字，文字列型とも read 文や write 文で入出力可能である．つまりキーボードからデータを読み込んだりディスプレイに表示したりできる．

文字列の型名は string であり，配列と同じように [ ] で括って最大文字数を前もって決めておく．[ ] 内の数値により，何番目の文字かを示すことができる．最大値は 255 である．

```
program mojiretu(input,output);
var st:string[80];
begin
  readln(st);       { キーボード入力 }
  st:='abcdefg';         { 代入例 }
  st:='日本語の文章'; { 漢字代入例 }
  st:=st+'XYZ'+st;    { 文字列の結合 }
  writeln(st)         { 表示 }
end.
```

文字列データは，文字データ同様文字列を引用符で括って表わす．

st：＝'abcdefg'；

が実行されると，st[0]に文字数が，st[1]以降に文字の文字コードが代入される．$61，$62，……は前の文字コード表から'a'，'b'，……であることがわかる．

st[0]	st[1]	st[2]	st[3]	st[4]	st[5]	st[6]	st[7]
7	$61	$62	$63	$64	$65	$66	$67

ただし0番目の文字数の数値を取り出すにはord(st[0])とするか，length(st)という関数を用いる．

st：＝'日本語の文章'；

が実行されると，次のように代入される．$93，$FAは2つで'日'，$96，$7Bも2つで'本'という漢字を表す．これは多くのパソコンで使われているシフトJISコードで漢字を表した結果である．

st[0]	st[1]	st[2]	st[3]	st[4]	st[5]	st[6]	st[7]
12	$93	$FA	$96	$7B	$8C	$EA	・・・

全角文字の場合，1つの文字が半角文字2文字分を占める．したがって，6文字でも配列の占める大きさは12になる．

また文字や文字列を＋で結んで，つなぐこともできる．文字列変数に文字列を代入する場合は宣言した範囲を超えてはならない．

[例題 3.13] 英単語への分解

英語の文章を入力し，英単語に分解する．

プログラム 3.13

```
{  *** 英単語への分解 ***  }
program prog3_13(input,output);
const sp = ' ';
var st:string[80];
    n, i, word, start : integer;
begin
  writeln('英文を入力して下さい ?');
  readln(st);
  n := length(st);
  i := 0;
  word := 0;
  repeat
```

```
    repeat i:=i+1 until (i=n) or (st[i]<>sp);
    start := i;
    repeat i:=i+1 until (i>n) or (st[i]=sp);
    write('頭文字= ',upcase(st[start]),' ':5);
    writeln(copy(st,start,i-start));
    word := word+1
  until i >= n;
  writeln('文字数= ',n,'  単語数= ',word)
end.
```

◆実行例

```
英文を入力して下さい ?
The marvellous thing is that it's painless.
頭文字= T     The
頭文字= M     marvellous
頭文字= T     thing
頭文字= I     is
頭文字= T     that
頭文字= I     it's
頭文字= P     painless.
文字数= 43  単語数= 7
```

文字・文字列を取り扱う手続き・関数がたくさん用意されているので，それらが大いに役立つ．詳細は付録参照のこと．

(c) ファイルからの入力

ファイルには大きく分けて，テキストファイルとバイナリファイルの2種類がある．テキストファイルはエディタなどで作成した文字の集合ファイルである．例えばPascalの(ソース)プログラムもテキストファイルである．

Pascalプログラムを読み込んで画面に表示するプログラムは次のようになる(MS-DOSのTYPEコマンドに等しい)．4行目の' 'で囲まれた部分は実際に存在するファイルの名前を指定する．

```
 1  program fileRead(input,output);
 2  var df:text;  st:string[120];
 3  begin
 4    assign(df,'PROG.PAS');  { ファイル名の指定 }
 5    reset(df);
 6    while not eof(df) do begin
 7      readln(df,st);           { 1行入力 }
 8      writeln(st)              { 表示 }
 9    end;
10    close(df)
11  end.
```

ファイルとのやりとりは，テキスト型のファイル変数 df を用いて行なう．すなわち df はユーザがつけた変数名である．assign 手続きは，ファイル変数 df とファイル名とを結びつける役割を果たす．次の reset はファイルからデータを入力するための準備をする手続きである．したがって，指示されたファイルが存在しない場合にはエラーとなる．

テキストファイルから入力を行う場合，先頭から順にデータを読んでいく．したがって最後はデータの終わりとなって入力できなくなる．これをチェックするのが関数 eof である．データが読めるとき false，もうデータがないとき true の値を返す．

実際にデータの読み込みを行なうのは，手続き readln, read である．これはいままで扱った read/readln と同じであり，先頭の引数の df を省略するとコンソール（キーボード）から入力し，df を指定するとそのファイル変数に対する入力を行なう．最長で 255 文字までで，ワープロで作成した改行の入っていない長い文字列を読み込むことはできない．

最後に，ファイルに対するアクセスを終了するという意味で手続き close を呼ぶ．

ファイルに書き込む場合は，reset の代わりに rewrite，readln (df, ...) の代わりに writeln (df, ...) を用いる．

```
program fileWrite(input,output);
var df:text;
    i:integer;
begin
  assign(df,'SQRT.TXT');   { ファイル名の指定 }
  rewrite(df);             { 新規作成 }
  writeln('20までの平方根表');
  for i:=1 to 20 do
    writeln(df,i:3,' : ',sqrt(i):10:8);
  close(df)
end.
```

同じ名前のファイルが存在すると，新しい内容に更新されるので注意が必要である．この他ファイル関係の関数・手続きには次のようなものがある．

- Append (var f : text)　既存のファイルを追加モードでオープンする．rewrite の代りに用いる．
- Flush (var f : text)　出力用にオープンされているセクタバッファをファイルに書き込んで空にする．
- b := Eoln (f)　行の終わりの判定
- b := SeekEof (f)　ブランク，タブをスキップしたファイルの終わりの判定
- b := SeekEoln (f)　ブランク，タブをスキップした行の終わりの判定
- w := IOResult　最後に行われた I/O 操作の状態を示す値を返す．結果は word 型．

結果が0は正常だったことを意味する．なお，デフォルトは{$I+}なので，{$I−}にしておく必要がある．

［例題 3.14］ 文字列の探索

ファイルから文章を読み込んで，文字列を探索する．

プログラム 3.14

```
{ *** 文字列の探索 *** }
program prog3_14(input,output);
var df:text;  i:integer;
    name,st,ser:string[80];
begin
  write('ファイル名 ? ');
  readln(name);
  assign(df,name);
  repeat
    reset(df);
    write('探索する文字列 ? ');
    readln(ser);
    if ser='' then begin   { 空文字 }
      close(df);
      halt                  { 終了 }
    end;
    i:=0;
    while not eof(df) do begin
      i:=i+1;
      readln(df,st);
      if pos(ser,st)>0 then
        writeln(i:3,':',st)
    end
  until false
end.
```

このプログラムでは，ファイルの名前，読み込んだ文字列，探索する文字列の3つに文字列変数を用いている．

◆実行例

```
ファイル名 ? enews.txt↵
探索する文字列 ? Kaifu↵
115: Prime Minister Toshiki Kaifu is planning to appear on a U-S T-V
121: during the Gulf war. He said Mr Kaifu thinks it very important
123: said Mr Kaifu wants to appear on a major U-S T-V network program
```

```
探索する文字列 ? Gorb⏎
200: President Gorbachev has expressed his hope that the world's
203: Mr Gorbachev said that his invitation to address the G-7 leaders
209: Mr Gorbachev has indicated willingness to implement radical
214: President Gorbachev's Union Treaty, which establishes a new
221: was drafted by Mr Gorbachev and the parliamentary chiefs of 9 of
探索する文字列 ? ⏎
```

このようなファイルがない場合は，プログラムのファイル名を指定してbeginやendなどの予約語を探してみるとよい．

(d)　練習問題

[7-1]　整数n(longint型)を入力して金額を表示するときのように，3桁ごとにコンマを打って表示しなさい．これは以前出題した問題であるが，文字列を使って作成してみなさい．

例：　12345678　→　12,345,678

[7-2]　例題3.14で使った英文ファイルを入力し，アルファベットの使用頻度を調べなさい．

[7-3]　テキストファイルの文字数と行数を数えて表示しなさい．できたら半角文字と全角文字を分けて数えなさい．

[7-4]　ある大学では6桁の学生証番号，たとえばこれを123456とすると，(1, 2, 3, 4, 5, 6)という6次元ベクトルと(1, 3, 1, 3, 1, 3)という定ベクトルの内積の下1桁の1, 2, 3～, 9, 0に対してA～Jの記号を割り当てる．いろいろな学生証番号を入力してそれに対する記号を求めなさい．

[7-5]　整数n(n > 0)を入力して漢数字で表示しなさい．

例：　1234　→　千二百三十四

[7-6]　整数nを16進法で表示しなさい．10～15にA～Fの文字を割り当てることにする．

[7-7]　Zellerの公式で年月日に対する曜日をローマ字または漢字で表示しなさい．年＋(13＊月＋8) div 5＋日を7で割った余りに曜日を対応させるのがZellerの公式である．1，2月は前年の13，14月とし，閏年は年に1を加える．0～6が日曜～土曜に対応する．

[7-8]　縦10，横に50個の0～9の数字をランダムに並べた乱数表を10個作り，それぞれをRANDOM. 001～RANDOM. 010という名前のファイルに格納しなさい．

[7-9]　上記のファイルを全部読んで0～9の数字がそれぞれいくつあったか，カウントしなさい．

[7-10]　4を整数の和で表すと，1＋1＋1＋1，1＋1＋2，1＋3，2＋2という組み合わせがある．整数n(n > 0)に対してすべての和の組み合わせを求めなさい．

第II部　応用　編

第4章 基本算法

これまでにも数値計算の例題をいくつか示したが，ここでは科学・技術の分野でしばしば用いられる各種数値計算法のなかでも，特に基本的かつ代表的な算法を，その初歩的なものに限って数種類紹介する．これらの算法は，計算機の高速な演算処理能力を利用した ⓘ 反復繰返し計算による求解(4.1節)，ⓘⓘ 連続量の離散化による近似計算(4.2および4.5節)，ⓘⓘⓘ ベクトルやマトリックスなど，多数の元に対する機械的計算処理(4.3および4.4節)などの点にそれぞれ特色がある．このような数値計算においては，反復法の収束性，離散化による誤差の程度の見積りのほか丸め誤差や桁おち等の問題にも留意する必要があるが，本書では省略した．詳しくは数値計算法に関する専門書を参照されたい．

4.1 Newton-Raphson 法

高次の代数方程式あるいは，超越方程式の根を求めるには逐次近似を行なう．この場合，適当な推定値から出発して，繰返し計算によって正しい解に系統的に近づくことをめざすのであるが，その方法の一つに Newton-Raphson 法がある．

$f(x)=0$ の根 x の近似値を x_k とし，Taylor 級数展開の x に関して1次までの項によって $f(x)$ を近似すると，

$$f(x) \fallingdotseq f(x_k)+f'(x_k)(x-x_k)=0 \qquad \cdots\cdots(4.1)$$

を得る．x について，(4.1)を解いて，x の近似値 x_{k+1} が得られる．

$$x_{k+1}=x_k-f(x_k)/f'(x_k) \qquad (k=0,\ 1,\ 2,\ \cdots) \qquad \cdots\cdots(4.2)$$

反復公式(4.2)を用いて，$x=x_k$ における $f(x)$ の接線と x 軸との交点 x_{k+1} を計算して必要な精度まで根を改良していくことができる．

Newton-Raphson 法は，もし $f(x_k)\neq 0$ かつ $f'(x_k)=0$ の場合はうまくいかない．この場合は x_0 を選び直さなければならない．

この方法はまた複素数を根とする場合，あるいは方程式系にも拡張できる．

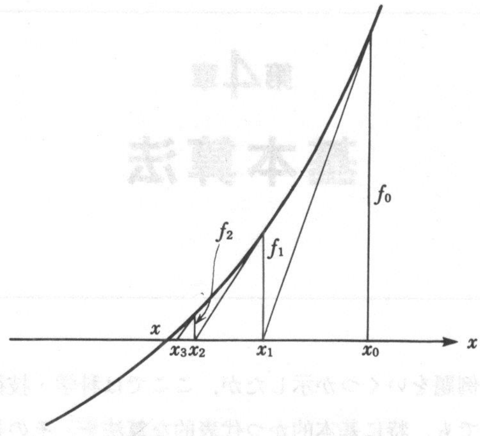


図 4.1 Newton-Raphson 法の概念図

［例題 4.1］ 3 次方程式の根

Newton-Raphson 法を用いて $x^3-a=0$ の根を求める．

プログラム 4.1

```
program newton; {  *** ニュートン法 ***  }
var a : real;

function f(x : real) : real;  {  関数f(x)  }
begin
  f := x * x * x - a;
end;

function df(x : real) : real;  {  導関数f'(x)  }
begin
  df := 3.0 * x * x;
end;

const eps = 1.e-8;
      max = 100;
var i : integer;
    x, dx : real;

begin
  write(' a, x (初期値) ? ');
  readln(a,x);
  writeln('i':3,'x':8,'f(x)':16,'df(x)':14);
  i := 0;
  repeat
    dx := -f(x) / df(x);
    x := x + dx;
    writeln(i:3,'  ',x:12,'  ',f(x):12,'  ',df(x):12);
    i := i + 1;
  until (i >=max) or (abs(dx) < eps);
  writeln('解＝ ',x:12);
end.
```

〈実行例〉

```
a, x (初期値) ? 2 1
  i      x              f(x)            df(x)
  0  1.333333E+00  3.703704E-01  5.333333E+00
  1  1.263889E+00  1.895523E-02  4.792245E+00
  2  1.259933E+00  5.925932E-05  4.762297E+00
  3  1.259921E+00  5.893526E-10  4.762203E+00
  4  1.259921E+00  0.000000E+00  4.762203E+00
解=  1.259921E+00
```

4.2　数値積分

x の関数 $f(x)$ の区間 $[a, b]$ にわたる定積分 $I=\int_a^b f(x)\,dx$ を考える．$f(x)$ が複雑な関数で，その不定積分（原始関数）を求めることが困難な場合，この定積分を近似的に求める方法が必要となる．ここで $f(x)$ は区間 $[a, b]$ で連続で，その不定積分はわからないけれども，$f(x)$ の値そのものは $[a, b]$ 内のどこででも求めうるものとする．

この定積分を求める最も素直な方法は，関数 $f(x)$ を別の積分可能な関数 $p(x)$ で置き換えることによって近似積分を求めることであろう．この $f(x)$ の近似関数 $p(x)$ として，多項式を用いることにすると，Newton-Cotes の公式と総称されうるさまざまな数値積分の公式が導かれる．

最も単純な $p(x)$ としては，積分区間の両端点 $(a, f(a))$ および $(b, f(b))$ を通る直線，すなわち1次の多項式で $f(x)$ を近似するものがあり，これは台形公式と呼ばれる．当然，積分区間が広くなれば誤差が大きくなるから，もとの積分区間を多数の微小区間に分割し，各微小区間毎に区分積分を行う．

もう一段高次の近似では，$p(x)$ を区間の両端 $x=a$，$x=b$ とその中間点 $x=(a+b)/2$ で $f(x)$ に一致するような2次の多項式とするもので，これによる近似積分はシンプソン則による積分と呼ばれる．この場合も微小区間にわけて区分積分を行うが，微小区間数は偶数個でなければならない．

(a)　台形公式による方法

数値積分の中で最も単純なものは，積分区間を等分して各微小区間を台形とみなし，その面積の和として求めるものである．

微小区間の面積は

$$\Delta S=\frac{(f(x_i)+f(x_{i+1}))\cdot\Delta x}{2}$$

であるので，積分区間 $[a, b]$ での積分値は，この区間を n 等分し，$x_0=a$，$x_n=b$ として，微小区間毎に区分積分を行なうことにすれば

$$\int_a^b f(x)\,dx \fallingdotseq S=\left(\frac{f(a)+f(b)}{2}+\sum_{i=1}^{n-1} f(x_i)\right)\cdot\Delta x$$

となる．

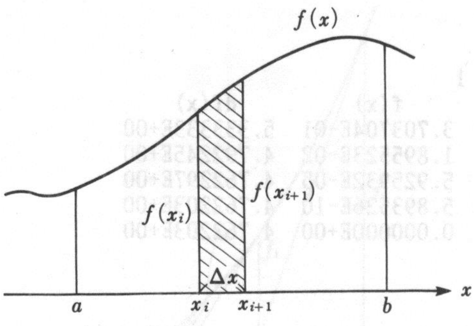


図 4.2 台形公式による数値積分

[例題 4.2] 台形公式による数値積分

$\int_0^1 \frac{1}{1+x^2}\,dx\left(=\frac{\pi}{4}\right)$ を台形公式により求める．

プログラム 4.2

```
program trapezoid; {  *** 台形公式による積分 ***  }

function f(x : real) : real;    {  関数f(x)  }
begin
  f := 1.0 / (1.0 + x * x);
end;

var i, n[illegible] : integer;
    sum, x, dx, xs, xe : real;

begin
  write('分割数 ? ');
  readln(n[illegible]);
  xs := 0.0;
  xe := 1.0;
  sum := (f(xs) + f(xe)) / 2.0;
  dx := (xe - xs) / n[illegible];
  for i := 1 to n[illegible] - 1 do begin
    x := xs + dx * i;
    sum := sum + f(x);
  end;
  sum := sum * dx;
  writeln('結果  =',sum:12:9);
  writeln('真の値=',pi/4:12:9);
end.
```

〈実 行 例〉

```
分割数 ? 10
結果  = 0.784981497
真の値= 0.785398163
```

(b) シンプソンの1/3則による方法

$f(x)$を2次式で近似した場合の公式は

$$\int_a^b f(x)\,dx \doteqdot \frac{b-a}{6}\left[f(a)+f(b)+4f\left(\frac{a+b}{2}\right)\right]$$

である．これをシンプソンの公式(シンプソンの$\frac{1}{3}$則)という．これは区間$[a, b]$において

$$p(a)=f(a) \quad , \quad p(b)=f(b) \quad , \quad p\left(\frac{a+b}{2}\right)=f\left(\frac{a+b}{2}\right)$$

となる2次式$p(x)$で$f(x)$を近似した場合の積分公式である．

区間$[a, b]$を$2n$等分したうえで2微小区間毎に区分積分を施していくと，$x_0=a$, $x_{2n}=b$として，

$$\int_a^b f(x)\,dx \doteqdot \frac{\Delta x}{3}\Big(f(a)+f(b)\Big)+\frac{2\Delta x}{3}\sum_{j=1}^{n-1} f(x_{2j})+\frac{4\Delta x}{3}\sum_{j=1}^{n} f(x_{2j-1})$$

という近似式が導かれる．

[例題4.3] シンプソン公式による数値積分

$\int_0^1 \frac{1}{1+x^2}\,dx\left(=\frac{\pi}{4}\right)$をシンプソン公式により求める．

プログラム4.3

```
program simpson; {  *** シンプソン公式による積分 ***  }

function f(x : real) : real;     {  関数f(x)  }
begin
  f := 1.0 / (1.0 + x * x);
end;

var i, ns : integer;
    sum, sumo, sume, x, dx, xs, xe : real;

begin
  write('分割数 ? ');
  readln(ns);
  xs := 0.0;
  xe := 1.0;
  sum := f(xs) + f(xe);
  sumo := 0.0;
  sume := 0.0;
  dx := (xe - xs) / (2.0 * ns);
  x := xs;
  for i := 1 to ns - 1 do begin
    x := x + dx;
    sumo := sumo + f(x);
```

```
      x := x + dx;
      sume := sume + f(x);
    end;
    x := x + dx;
    sumo := sumo + f(x);
    sum := (sum + 4.0 * sumo + 2.0 * sume)
               * dx / 3.0;
    writeln('結果  =',sum:12:9);
    writeln('真の値=',pi/4:12:9);
  end.
```

〈実 行 例〉

```
分割数 ? 10
結果  = 0.785398163
真の値= 0.785398163
```

4.3 連立 1 次方程式の解法

小規模な連立 1 次方程式は消去法によって解かれる．より大規模な連立 1 次方程式(おおむね数十元以上)の場合には，繰返し算法の一種である Gauss-Seidel の反復法や緩和法が用いられる．ここでは我々が筆算で行なうのと同様の消去法を示す．

消去法とは次の 3 つの演算を適当に何回か行なうことによって，未知数を消去し，与えられた方程式をより簡単な方程式に変形して解を求める方法である．

(1) 一つの方程式に 0 でない数を掛ける．

(2) 一つの方程式にある数を掛けて他の方程式に加える．

(3) 二つの方程式を入れ替える．

たいていの消去法は，有名な Gauss の消去法を含めて，未知数を 1 回に 1 個ずつ消去することによって係数行列を三角行列に変形して，次に後退置換によって未知数の値を計算する．

3 元連立 1 次方程式の場合を例にとって Gauss の消去法のプロセスを示せば次のようになる．ここに左側の 3 行 3 列は係数行列，第 4 列は定数ベクトルを表わす．

(**Gauss** の消去法)

1	1	−1	2	r_1
3	5	−7	0	r_2
2	−3	1	5	r_3
1	1	−1	2	r_1
0	2	−4	−6	r_2-3r_1
0	−5	3	1	r_3-2r_1

1	1	−1	2	r_1
0	1	−2	−3	$r_2/2$
0	0	−7	−14	$r_3+5/2r_2$
1	1	−1	2	r_1
0	1	−2	−3	r_2
0	0	1	2	$r_3/-7$

Gauss-Jordan 法は，第 n 列で n 行目以外の成分を0にする操作を繰り返して，係数行列を単位行列に変形する．したがって後退置換は不要となる．

（**Gauss-Jordan 法**）

1	1	−1	2	r_1
3	5	−7	0	r_2
2	−3	1	5	r_3
1	1	−1	2	r_1
0	2	−4	−6	r_2-3r_1
0	−5	3	1	r_3-2r_1
1	0	1	5	$r_1-1/2r_2$
0	1	−2	−3	$1/2r_2$
0	0	−7	−14	$r_3+5/2r_2$
1	0	0	3	$r_1+r_3/7$
0	1	0	1	$r_2-2r_3/7$
0	0	1	2	$r_3/-7$

実際のプログラムでは，第 n 列目で n 行目以外の成分を0にする際，第 n 列目で係数が最大のものを選び，行を入れ替えてそれ以外を0にするという操作(pivoting)をする．これは計算上の誤差を最小にするためである．

[例題 4.4]　連立 1 次方程式の解

Gauss-Jordan 法により n 元連立 1 次方程式を解く．

プログラム 4.4

```
program linear_eq;  {  *** 連立１次方程式 ***  }

type row = 1..20; col = 1..21;
     matrix = array [row,col] of real;
      line = array [row] of row;

var a : matrix;
     i, j, m, n : integer;
     dfile : text;


procedure sweep(var a : matrix; maxrow, maxcol : integer);
var v : matrix;
     pivo : line;
     i, j, p, temp, max : integer;
     app, aip : real;

begin
  for i := 1 to maxrow do    {  初期設定  }
    pivo[i] := i;
  for p := 1 to maxrow do begin
    max := p;               {  係数の最大値の探索  }
    for i := p + 1 to maxrow do
      if abs(a[pivo[  i],p]) >
         abs(a[pivo[max],p]) then max := i;
    temp := pivo[p];              {  データ交換  }
    pivo[p] := pivo[max];
    pivo[max] := temp;
    app := a[pivo[p],p];   {  １方程式の係数の除算  }
    for j := 1 to maxcol do
      a[pivo[p],j] := a[pivo[p],j] / app;
    for i := 1 to maxrow do   {  方程式どうしの加算  }
      if i <> p then begin
        aip := a[pivo[i],p];
        for j := 1 to maxcol do
          a[pivo[i],j] := a[pivo[i],j] - aip * a[pivo[p],j];
      end;
  end;
  for i := 1 to maxrow do      {  データ転送  }
    for j := 1 to maxcol do
      v[i,j] := a[i,j];
  for i := 1 to maxrow do
    for j := 1 to maxcol do
      a[i,j] := v[pivo[i],j];
end;

procedure write_matrix(var a : matrix; m, n : integer);
var i, j : integer;
```

```
begin
  writeln;
  for i := 1 to m do begin
    for j := 1 to n do
      write(a[i, j]:10:5);
    writeln;
  end;
end;

begin                     { メインプログラム }
  assign(dfile, 'LINEAR.DAT');
  reset(dfile);
  readln(dfile, m, n);
  for i := 1 to m do
    for j := 1 to n do
      read(dfile, a[i, j]);   { 行列要素の読み込み }
  write_matrix(a, m, n);
  sweep(a, m, n);
  write_matrix(a, m, n);
end.
```

〈LINEAR. DAT の例〉

```
3  4
1  1 -1  2
3  5 -7  0
2 -3  1  5
```

〈実 行 例〉

```
1.00000   1.00000  -1.00000   2.00000
3.00000   5.00000  -7.00000   0.00000
2.00000  -3.00000   1.00000   5.00000

1.00000   0.00000   0.00000   3.00000
0.00000   1.00000   0.00000   1.00000
0.00000   0.00000   1.00000   2.00000
```

行列の大きさや各成分の値はLINEAR. DATというテキストファイルから読み込む．配列pivoには何行目を入れ替えたかを示すデータが入っている．このプログラムでは，不定解，解のない場合に対して例外処理をしていないので，そのときは実行時にエラーとなる．

次のプログラムは連立1次方程式の解法を利用して，n 個の点を通る最小次数の多項式を求めるものである．ただし $x_i \neq x_j (i \neq j)$ とする．

[例題 4.5]　n 点を通る $n-1$ 次式

$f(x) = x^3 - 2x^2 - 11x + 12$ の $f(-1)$, $f(0)$, $f(2)$, $f(3)$ を与えてもとの方程式を得る．

プログラム 4.5

```
program curve;  { *** n点を通る n-1 次式 *** }

type row = 1..20; col = 1..21;
     matrix = array [row,col] of real;
     line = array [row] of row;

var a : matrix;
    i, j, n : integer;
    x : real;
    dfile : text;

procedure sweep(var a : matrix; maxrow, maxcol : integer);
var v : matrix;
    pivo : line;
    i, j, p, temp, max : integer;
    app, aip : real;

begin
  for i := 1 to maxrow do    { 初期設定 }
    pivo[i] := i;
  for p := 1 to maxrow do begin
    max := p;              { 係数の最大値の探索 }
    for i := p + 1 to maxrow do
      if abs(a[pivo[  i],p]) >
         abs(a[pivo[max],p]) then max := i;
    temp := pivo[p];               { データ交換 }
    pivo[p] := pivo[max];
    pivo[max] := temp;
    app := a[pivo[p],p];    { 1方程式の係数の除算 }
    for j := 1 to maxcol do
      a[pivo[p],j] := a[pivo[p],j] / app;
    for i := 1 to maxrow do    { 方程式どうしの加算 }
      if i <> p then begin
        aip := a[pivo[i],p];
        for j := 1 to maxcol do
          a[pivo[i],j] := a[pivo[i],j] - aip * a[pivo[p],j];
      end;
  end;
  for i := 1 to maxrow do      { データ転送 }
    for j := 1 to maxcol do
      v[i,j] := a[i,j];
  for i := 1 to maxrow do
    for j := 1 to maxcol do
      a[i,j] := v[pivo[i],j];
 end;

 function power(x : real; n : integer) : real;
 var i : integer;
     px : real;
 begin                     { x(実数型)のn乗 }
   px := 1.0;
   if n > 0 then
     for i := 1 to n do px := px * x;
   power := px;
 end;
```

```
begin                        {  メインプログラム  }
  assign(dfile, 'CURVE.DAT');
  reset(dfile);
  readln(dfile,n);
  for i := 1 to n do begin
    readln(dfile,x,a[i,n + 1]);    {  x,f(x)の読み込み  }
    for j := 1 to n do
      a[i,j] := power(x,n - j);    {  xの n-j 乗  }
    for j := 1 to n + 1 do
      write(a[i,j]:12:5);
    writeln;
  end;
  sweep(a,n,n + 1);            {  f(x)の係数の計算  }
  writeln;
  for i := 1 to n do
    writeln(n - i:5,a[i,n + 1]:10:5);
end.
```

〈CURVE. DAT 例〉

```
 4
-1  20
 0  12
 2 -10
 3 -12
```

〈実 行 例〉

```
-1.00000     1.00000    -1.00000     1.00000    20.00000
 0.00000     0.00000     0.00000     1.00000    12.00000
 8.00000     4.00000     2.00000     1.00000   -10.00000
27.00000     9.00000     3.00000     1.00000   -12.00000

3    1.00000
2   -2.00000
1  -11.00000
0   12.00000
```

このプログラムの手続き sweep は前の例と全く同じである．関数 power は実数 x の n 乗の計算を行なう．

4.4 逆　行　列

逆行列の計算にもやはり Gauss-Jordan 法を用いることができる．対象とする行列を単位行列に変形するのと全く同じ操作を単位行列に対して加えると逆行列が求まる．一例を示すと下の表のようになる．

r_1	1	3	2	1	0	0
r_2	−1	−2	−1	0	1	0
r_3	2	4	3	0	0	1

r_1	1	3	2	1	0	0
r_2+r_1	0	1	1	1	1	0
r_3-2r_1	0	−2	−1	−2	0	1
r_1-3r_2	1	0	−1	−2	−3	0
r_2	0	1	1	1	1	0
r_3+2r_2	0	0	1	0	2	1
r_1+r_3	1	0	0	−2	−1	1
r_2-r_3	0	1	0	1	−1	−1
r_3	0	0	1	0	2	1

[例題 4.6]　*n* 次正方行列の逆行列

Gauss-Jordan 法により *n* 次の逆行列を求める.

プログラム 4.6

```
program inversion_matrix;  {  *** 逆行列 ***  }

type matrix = array [1..20, 1..20] of real;
     vector = array [1..20] of real;

{  Gauss-Jordan法によるｎ次行列ａの逆行列  }
procedure gau_jor(var a : matrix; n : integer);
var e : matrix;
    pivo : vector;
    iok, i, j, k, lh, l : integer;
    h, max : real;

begin
  for i := 1 to n do
    pivo[i] := 0;
  for j := 1 to n do
    for i := 1 to n do
      if i = j then e[j,i] := 1.0
               else e[j,i] := 0.0; {  単位行列ｅ  }
  for k := 1 to n do begin
    max := 0.0;      {  ｉ 列目が最大の行(ｊ 行)を探し出す  }
    for i := 1 to n do begin
      h := abs(a[i,k]);
      if h > max then
        if pivo[i] = 0 then begin
          max := h;
          j := i
        end
```

```
    end;
    pivo[j] := k; {  行の入れ替えを記録する  }
    h := 1.0 / a[j,k];
    for lh := 1 to n do begin
      a[j,lh] := a[j,lh] * h;
      e[j,lh] := e[j,lh] * h;  {  行列aとeに同じ操作  }
    end;
    for i := 1 to n do
      if i <> j then begin
        h := a[i,k];
        for lh := 1 to n do begin
          a[i,lh] := a[i,lh] - a[j,lh] * h;
          e[i,lh] := e[i,lh] - e[j,lh] * h;
        end;
      end;
  end;
   for k := 1 to n do begin
     for i := 1 to n do
       if pivo[i] = k then l := i;
     for lh := 1 to n do
       a[k,lh] := e[l,lh];
   end;
  end;

  var a : matrix;
      n, i, j : integer;
      dfile : text;

  begin                      {  メインプログラム  }
    assign(dfile, 'MATRIX.DAT');
    reset(dfile);
    readln(dfile, n);
    for i := 1 to n do
      for j := 1 to n do
        read(dfile, a[i,j]);
    gau_jor(a,n);
    for i := 1 to n do begin
      for j := 1 to n do write(a[i,j]:10:5);
      writeln;
    end;
  end.
```

〈MATRIX. DAT の例〉

```
 3
 1  3  2
-1 -2 -1
 2  4  3
```

〈実 行 例〉

```
-2.00000  -1.00000   1.00000
 1.00000  -1.00000  -1.00000
 0.00000   2.00000   1.00000
```

このプログラムでは a がもとの行列，e が単位行列である．上述の規則により a を単位行列に変換していくと，e が逆行列になる．行列が正則でない場合の判断は，pivoting の操作を行なう際，対角成分の絶対値最大の要素が十分小さい値(たとえば 10^{-5})であるかどうかでわかる．

4.5 微分方程式

高階(n 階)の常微分方程式は一般に n 組の連立 1 階常微分方程式に帰着できる．したがって，常微分方程式の初期値問題の数値解法としては，1 階の微分方程式に関する解法が基本となる．この場合にも数値積分の場合と同様に，解を求めようとする独立変数の区間を微小な区間(Δx)に等分割し，$x_i = x_0 + i\Delta x,\ i = 1, 2, 3, \cdots$ における $y(x_i)$ の近似値 y_i を順次求めていく方法がとられる．

(a) Euler 法

問題

$$y' = f(x, y) \quad , \quad y(x_0) = y_0 \qquad \cdots\cdots(4.3)$$

を与えられると，x_i における y の値 y_i がわかれば x_i における y の微係数 y_i' は(4.3)によって $f_i \equiv f(x_i, y_i)$ と計算できる．したがって x_i から x_{i+1} までの間の微小区間についての積分は

$$\Delta y_i = y_{i+1} - y_i = \int_{x_i}^{x_{i+1}} f(x, y)\,dx \fallingdotseq (x_{i+1} - x_i) f_i$$

によって近似することができる．

これを y_{i+1} について解いて

$$y_{i+1} \fallingdotseq y_i + (x_{i+1} - x_i) f_i$$

が得られる．これは Euler の公式として知られている．あまり精度が良くないので，Euler 法がそのままの形で用いられることは稀であるが，基本的な考え方がわかりやすいので例示した．

[例題 4.7] Euler 法による微分方程式

$y' = x + y$ を Euler 法で解き，真の解と比較する．

プログラム 4.7

```
program euler_method;  {  *** オイラー法 ***  }

function dy_dx(x, y : real) : real;  {  dy/dx = x + y  }
begin
  dy_dx := x + y;
end;

procedure euler(var y : real; x, step : real);
```

```
begin
  y := y + dy_dx(x,y) * step;
end;

const interval = 10;
var x, y, xinit, yinit, xend, step, strict : real;
    count : integer;

begin
  write('x, y の初期値 ? ');
  readln(xinit,yinit);
  write('x の終値と幅 ? ');
  readln(xend,step);
  count := 0;
  x := xinit;
  y := yinit;
  writeln(' x :',x:12, '    y : ',y:12);
  repeat
    count := count + 1;
    euler(y,x,step);
    x := x + step;
    if count = interval then begin
      count := 0;
      write(' x :',x:12, '    y : ',y:12);
      strict := (1.0 + xinit + yinit) *
                  exp(-xinit) * exp(x) - 1.0 - x;
      writeln('  strict y : ',strict:12);
    end;
  until x >= xend;
end.
```

〈実 行 例〉

```
x, y の初期値 ? 0 1
x の終値と幅 ? 1 0.01
 x :0.000000E+00    y : 1.000000E+00
 x :1.000000E-01    y : 1.109244E+00  strict y : 1.110342E+00
 x :2.000000E-01    y : 1.240380E+00  strict y : 1.242806E+00
 x :3.000000E-01    y : 1.395698E+00  strict y : 1.399718E+00
 x :4.000000E-01    y : 1.577727E+00  strict y : 1.583649E+00
 x :5.000000E-01    y : 1.789264E+00  strict y : 1.797443E+00
 x :6.000000E-01    y : 2.033393E+00  strict y : 2.044238E+00
 x :7.000000E-01    y : 2.313527E+00  strict y : 2.327505E+00
 x :8.000000E-01    y : 2.633430E+00  strict y : 2.651082E+00
 x :9.000000E-01    y : 2.997265E+00  strict y : 3.019206E+00
 x :1.000000E+00    y : 3.409628E+00  strict y : 3.436564E+00
```

(b) Runge-Kutta 法

Runge-Kutta 法は方程式 $y' = f(x, y)$ を与えられたとき,

$$y_{i+1} = y_i + (k_1 + 2k_2 + 2k_3 + k_4)/6$$

によって x_i と y_i から x_{i+1} における y_{i+1} を定める. ただし

$$k_1 = \Delta x \cdot f(x_i, y_i)$$

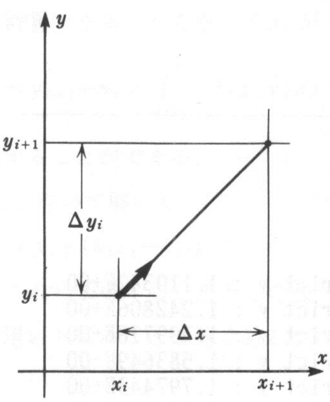
$$k_2 = \Delta x \cdot f\left(x_i + \frac{\Delta x}{2},\quad y_i + \frac{k_1}{2}\right)$$

$$k_3 = \Delta x \cdot f\left(x_i + \frac{\Delta x}{2},\quad y_i + \frac{k_2}{2}\right)$$

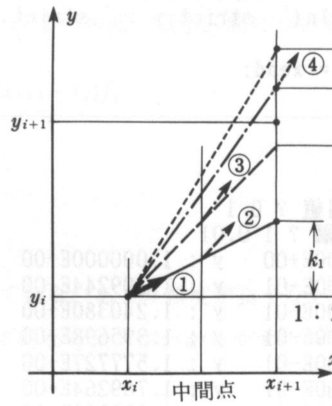
$$k_4 = \Delta x \cdot f(x_i + \Delta x, y_i + k_3)$$

である．これはくわしくは四次の Runge-Kutta 法といわれる．この方法は Euler 法より精度ははるかに良い．

図 4.3 は Euler 法および Runge-Kutta 法それぞれにおいて，x_i と y_i とから x_{i+1} における y_{i+1} を算出するプロセスを模式的に示したものである．Euler 法においては，点 (x_i, y_i) での微係数をそのまま x_{i+1} まで外挿して y_{i+1} を定めている．これに対して Runge-Kutta 法では (i) まず点 (x_i, y_i) における微係数(図中の実線矢印①)から y の増分 k_1 を推定する．次に，(ii) 点 (x_i, y_i) といま得た点 $(x_{i+1}, y_i + k_1)$ の中間点での微係数(破線矢印②)から，これに平行な破線で示す直線によって増分 k_2 の推定を行なう．(iii) これをさらに繰返して，(1点鎖線矢印③)から第3の増分推定 k_3 を得る．(iv) ついで第3の推定点 $(x_i + \Delta x, y_i + k_3)$ における微係数(点線矢印④)から第4の増分推定 k_4 を算出する．(v) そして最後にこれら4つの増分推定を 1：2：2：1 の重みをつけて平均化し，最終的な増分推定とする．



(a) Euler 法



(b) 4次の Runge–Kutta 法

図 4.3 Euler 法と Runge-Kutta 法の比較

[例題 4.8] Runge-Kutta 法による微分方程式の解

前の例題と同じ $y' = x + y$ を Runge-Kutta 法で解き，真の解と比較する．

プログラム 4.8

```
program runge_kutta_method;  { *** ルンゲクッタ法 *** }

function dy_dx(x, y : real) : real;  { dy/dx = x+y }
begin
  dy_dx := x + y;
end;

procedure runge_kutta(var y : real; x,step : real);
var dy1, dy2, dy3, dy4 : real;

begin
  dy1 := dy_dx(x,y)*step;
  dy2 := dy_dx(x + step*0.5, y + dy1*0.5)*step;
  dy3 := dy_dx(x + step*0.5, y + dy2*0.5)*step;
  dy4 := dy_dx(x + step     , y + dy3     )*step;
  y := y + (dy1 + 2.0*(dy2 + dy3) + dy4) / 6.0
end;

const interval = 10;
var x, xinit, xend, y, yinit, step, strict : real;
    count : integer;

begin
  write('x, y の初期値 ? ');
  readln(xinit,yinit);
  write('x の終値と幅 ? ');
  readln(xend,step);
  count := 0;
  x := xinit;
  y := yinit;
  writeln(' x :',x:12, '    y : ',y:12);
  repeat
    count := count + 1;
    runge_kutta(y,x,step);
    x := x + step;
    if count = interval then begin
      count := 0;
      write(' x :',x:12, '    y : ',y:12);
      strict := (1.0 + xinit + yinit) *
                exp(-xinit) * exp(x) - 1.0 - x;
      writeln('  strict y : ',strict:12);
    end;
  until x >= xend;
end.
```

〈実 行 例〉

```
x, y の初期値 ? 0 1
x の終値と幅 ? 1 0.01
 x :0.000000E+00    y : 1.000000E+00
 x :1.000000E-01    y : 1.110342E+00   strict y : 1.110342E+00
 x :2.000000E-01    y : 1.242806E+00   strict y : 1.242806E+00
 x :3.000000E-01    y : 1.399718E+00   strict y : 1.399718E+00
 x :4.000000E-01    y : 1.583649E+00   strict y : 1.583649E+00
 x :5.000000E-01    y : 1.797443E+00   strict y : 1.797443E+00
 x :6.000000E-01    y : 2.044238E+00   strict y : 2.044238E+00
 x :7.000000E-01    y : 2.327505E+00   strict y : 2.327505E+00
```

```
x :8.000000E-01   y : 2.651082E+00  strict y : 2.651082E+00
x :9.000000E-01   y : 3.019206E+00  strict y : 3.019206E+00
x :1.000000E+00   y : 3.436564E+00  strict y : 3.436564E+00
```

(c) 高階微分方程式

次の例は，高階微分方程式の数値解例として，2次元ケプラー運動の方程式(2つの2階微分方程式)をRunge-Kutta法で解いたものである．

いま，惑星の運動面内に太陽を原点とする直角座標系(x_1, x_2)をとると，適当に無次元化されたケプラー運動の方程式は

$$\frac{d^2 x_1}{dt^2} = -\frac{x_1}{r^3}, \qquad \frac{d^2 x_2}{dt^2} = -\frac{x_2}{r^3} \qquad \cdots\cdots(4.4)$$

と書ける．ここにrは太陽から惑星までの距離で$r=(x_1{}^2+x_2{}^2)^{1/2}$である．

ここで，

$$y_1 = x_1$$
$$y_2 = x_2$$
$$y_3 = dx_1/dt$$
$$y_4 = dx_2/dt$$

と置けば，(4.4)式は

$$\left.\begin{aligned} y_1' &= y_3 \\ y_2' &= y_4 \\ y_3' &= -y_1/r^3 \\ y_4' &= -y_2/r^3 \end{aligned}\right\} \qquad \cdots\cdots(4.5)$$

と変形でき，4元の連立1階微分方程式に帰着される．そこで，惑星の位置および速度の各成分に当たるy_1，y_2，y_3，y_4を成分とするベクトル$\boldsymbol{y}$を考えれば，(4.5)式の4つの微分方程式は

$$\boldsymbol{y}' = \boldsymbol{f}(\boldsymbol{y}) \qquad \cdots\cdots(4.6)$$

と書ける．ここに$\boldsymbol{f}$は関数ベクトルで

$$\boldsymbol{f}(\boldsymbol{y}) = \begin{pmatrix} y_3 \\ y_4 \\ -y_1/r^3 \\ -y_2/r^3 \end{pmatrix}$$

である．式(4.6)は4元の1階連立微分方程式であるから，4個の初期条件を与えれば，以後Runge-Kutta法をベクトル形に拡張することにより，順次積分していくことができる．

プログラムでは，変数xが無次元時間，yが上述の4次元ベクトル，dyがベクトルyの1階微分係数を表す．なお，ケプラー運動においては，角運動量(プログラム中では変数ang_mで表す)および全エネルギー(変数energyで表す)が保存される．数値解析が正しくかつ充分精度よく実行されていれば，どの時間ステップにおいても，これらが不変に保たれるはずである．プログラムで

はこれらの値をも計算・表示してチェックしている．

[例題 4.9]　2次元ケプラー運動

2次元ケプラー運動の微分方程式を Runge-Kutta 法で解く．

プログラム 4.9

```
program kepler; {  ***  2次元ケプラー運動 ***  }
type dep_var = array[1..4] of real;
const y : dep_var = (1, 0, 0.7071068, 0.7071068);

{  dy/dx( =f(x,y) )の計算  }
procedure dy_dx(x : real; var y, dy : dep_var);
var r, rcube  : real;
begin
  r := sqrt(y[1]*y[1] + y[2]*y[2]);
  rcube := r*r*r;
  dy[1] := y[3];
  dy[2] := y[4];
  dy[3] := -y[1]/rcube;
  dy[4] := -y[2]/rcube;
end;

{  ルンゲクッタ法による微分方程式の解  }
procedure runge_kutta(var y : dep_var; x, step : real);

  procedure vector_sum(var c, a, b : dep_var; s : real);
  var i : integer;
  begin
    for i := 1 to 4 do
  c[i] := a[i] + b[i] * s;
  end;

  procedure vector_scalar(var b, a : dep_var; c : real);
  var i : integer;
  begin
    for i := 1 to 4 do
      b[i] := a[i] * c;
  end;

var dy1, dy2, dy3, dy4, ydum, dydum : dep_var;
begin
  dy_dx(x,y,dydum);
  vector_scalar(dy1,dydum,step);

  vector_sum(ydum,y,dy1,0.5);
  dy_dx(x + step*0.5,ydum,dydum);
  vector_scalar(dy2,dydum,step);


  vector_sum(ydum,y,dy3,1.0);
  dy_dx(x + step,ydum,dydum);
  vector_scalar(dy4,dydum,step);
```

```
  vector_sum(dydum,dy1,dy2,2.0);
  vector_sum(dydum,dydum,dy3,2.0);
  vector_sum(dydum,dydum,dy4,1.0);
  vector_sum(y,y,dydum,1.0/6.0);
 end;

const interval = 10;
var x, xinit, xend, step,
    energy, ang_m : real;
    i, count : integer;
    outfile : text;
begin                          {  メインプログラム  }
  xinit := 0.0;
  xend := 2.0 * pi;
  step := xend / 100.0;
  assign(outfile,'KEPLER.DAT');
  rewrite(outfile);
  writeln;
  count := 0;
  x := xinit;
  write(' x :',x:10, '    y : ');
  for i := 1 to 4 do
    write(y[i]:10,'  ');
  writeln;
  writeln(outfile,x,' ',y[1],' ',y[2]);
  repeat
    count := count + 1;
    runge_kutta(y,x,step);
    x := x + step;
    writeln(outfile,x,' ',y[1],' ',y[2]);
    if count = interval then begin
      count := 0;
      write(' x :',x:10, '    y : ');
      for i := 1 to 4 do
        write(y[i]:10,'  ');
      writeln;
      ang_m := y[1]*y[4] - y[2]*y[3];
      energy := -1.0 / sqrt(y[1]*y[1] + y[2]*y[2]) +
                0.5 * (y[3]*y[3] + y[4]*y[4]);
      writeln(' ang. m :',ang_m:10,'  energy : ',energy:10);
    end;
      until x >= xend;
      close(outfile);
    end.
```

〈実 行 例〉

```
 x :0.0000E+00   y : 1.0000E+00  0.0000E+00  7.0711E-01  7.0711E-01
 x :6.2832E-01   y : 1.2892E+00  4.2689E-01  2.6256E-01  6.3542E-01
ang. m :7.0711E-01  energy : -5.000E-01
 x :1.2566E+00   y : 1.3660E+00  7.8995E-01  -8.568E-04  5.1714E-01
ang. m :7.0711E-01  energy : -5.000E-01
 x :1.8850E+00   y : 1.3028E+00  1.0738E+00  -1.924E-01  3.8419E-01
ang. m :7.0711E-01  energy : -5.000E-01
 x :2.5133E+00   y : 1.1316E+00  1.2692E+00  -3.485E-01  2.3402E-01
ang. m :7.0711E-01  energy : -5.000E-01
 x :3.1416E+00   y : 8.6892E-01  1.3617E+00  -4.851E-01  5.3643E-02
ang. m :7.0711E-01  energy : -5.000E-01
 x :3.7699E+00   y : 5.2485E-01  1.3244E+00  -6.076E-01  -1.861E-01
```

```
ang. m :7.0711E-01  energy : -5.000E-01
x :4.3982E+00   y : 1.1103E-01  1.0998E+00  -7.000E-01  -5.651E-01
ang. m :7.0711E-01  energy : -5.000E-01
x :5.0265E+00   y : -3.090E-01  5.2165E-01  -5.096E-01  -1.428E+00
ang. m :7.0711E-01  energy : -5.000E-01
x :5.6549E+00   y : 2.6460E-01  -3.638E-01  1.8512E+00  1.2651E-01
ang. m :7.0695E-01  energy : -5.015E-01
x :6.2832E+00   y : 9.9898E-01  1.9700E-03  7.0442E-01  7.0906E-01
ang. m :7.0694E-01  energy : -5.015E-01
```

図4.4は，結果を別のプログラムによって図示したもので，初期値は位置ベクトル $\boldsymbol{x} = (1, 0)$，初期速度ベクトル $\boldsymbol{v} = (0.707, 0.707)$ で $\Delta t = 0.02\pi$ として $0 \leq t \leq 2\pi$ で数値積分し，求まった $\boldsymbol{x}$ を平面上にプロットした．解がケプラー運動の理論通り，閉じた楕円になっていることがわかる．

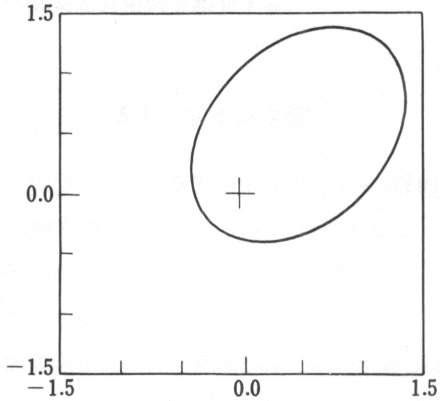


図 **4.4**　ケプラー運動の軌跡

```
[illegible]
    vector_sum(y,y,dydum,[illegible]
[illegible]
    vector_sum(y,y,dydum,[illegible]
[illegible]
var x, step, [illegible]
    energy, ang_m : real;
[illegible]
        step := [illegible]
[illegible]
        writeln;
        count := 0;
[illegible]
        for [illegible]
[illegible]
        writeln;
[illegible]
        repeat
            count := count [illegible]
[illegible]
            x := x + step;
[illegible]
            if count = int[illegible] then begin
                count := 0;
[illegible]
            end;
[illegible]
        until [illegible]
[illegible]
end.
```

[illegible]

第5章
データ構造の構築と利用

この章ではPascal特有のポインタ型，集合型，レコード型のデータの型について説明し，これらを用いた簡単なデータ構造の構築と利用例を紹介する．

5.1　ポインタ型

一般の変数は前もってデータの型や大きさを宣言しておくもの(静的変数)であるが，実行中に必要な型や大きさのデータの記憶域を確保したり，消去したりすることができる．この変数(動的変数)の記憶されている場所を指し示す値を持つのがポインタ型である．実際には，ポインタ型変数の値はメモリ空間上のアドレスである．

具体的に，ポインタ型や動的変数がどういう意味かは，次の例題がよく表わしている．

プログラム 5.1

```
program pointer_ex;
type p_to_integer = ^integer;
var ip : p_to_integer;

procedure inc2(var j : integer);
                  {  変数引数 : 整数  }
begin
  j := j + 2;
end;

procedure inc3(jp : p_to_integer);
                  {  値引数 : 整数へのポインタ  }
begin
  jp^ := jp^ + 3;
end;

begin
  new(ip);
  ip^ := 1;  write(ip^:5);
  inc2(ip^); write(ip^:5);
  inc3(ip);  writeln(ip^:5)
end.
```

〈実 行 例〉

1 3 6

ip は，整数へのポインタである．標準手続き new(ip) は，ip の指し示す場所を確保する手続きで，整数を指すときは整数の記憶域(2バイト)を，実数を指すときは実数の記憶域(6または8バイト)を，というようにポインタの指す型に合わせた記憶場所を確保する．ip^ なる表現は，ip の指し示すもの，すなわちこの例では整数型の変数である．

この例では，inc2 という手続きでは変数引数を使っているので，手続きの中でその変数の値を変えれば，もどったときに引き渡した変数の値は変化している．さて，inc3 では値引数ではあるが，それはポインタである．したがって，主プログラムから ip が手続き inc3 の引数 jp に渡されると，ip と jp が同じものを指すことになるため，jp^ を3増やせば，ip^ を3増やしたことになる．

このように，ポインタを用いると値引数を用いて変数引数と同じようなことをすることができる．変数引数というのは，実際には処理系内では変数のアドレスを手続きに渡しているわけで，プログラム作成者がそれを意識しなくてもよいわけである．

ポインタが指せるのは，ふつうの名前をつけて宣言される一般的な変数(静的変数)ではなく，標準手続き new によってプログラム中で作り出される動的変数である．手続き new は，引数のポインタ変数にそれが宣言されたときに決められた型の動的変数を1つ作りだし，それを指すポインタを引数へ代入する．なお，何も指していないことを表わすポインタ値として nil が用意されている．nil は，語記号で，すべてのポインタ型に共通である．

```
ip := nil;
```

を実行すると，ip は何も指していないことになる．

不要になった動的変数は，標準手続き dispose によって消すことができる．また，TURBO Pascal では mark と release という手続きで動的変数を解放することが多い．また memavail という関数で動的変数として使用可能なメモリ量を調べることもできる．

5.2 集 合 型

集合とは，同じ型のデータ(要素)を集めて，1つのまとまりとして扱うものである．

集合型の定義は，たとえば，

```
type name_of_the_week = (sunday, monday, tuesday, wednesday,
                         thursday, friday, saturday);
     dayoftheweek = set of name_of_the_week;
```

というように行なう．また次のような部分範囲型を用いて集合を定義することもできる．

```
type weekday = set of monday .. friday;
```

集合型の値は，もととなる型(基底型と呼ぶ)の値全体からなる集合の部分集合が値となり，各要

素をコンマで区切って全体を［　］で囲む．

他の構造型（配列・レコード）とは異なり，集合型には値の演算が定義されている．和・差・積の集合演算の結果は集合型であり，比較の結果は論理型になる．

和集合	+	[a, b, c] + [b, d, e] = [a, b, c, d, e]	
差集合	−	[a, b, c] − [b, d, e] = [a, c]	
積集合	*	[a, b, c] * [b, d, e] = [b]	
比較	=	[a, b, c] = [a .. c]	要素が等しい
	<>	[a, b, c] <> [a, c]	等しくない
	>=	[a, b, c, d, e] >= [a, b]	右辺が左辺にふくまれる
	<=	[d, e] <= [d .. f]	左辺が右辺にふくまれる
	in	f in [a..z]	左辺は右辺の要素である

ただしTURBO Pascalの集合型の制限として，要素は256個までである．したがってset of char, set of byteは可，set of integerは不可である．

［例題5.1］ 英文中の文字・母音

英文の中にどのような文字・母音字が含まれているかを調べる．

プログラム5.2

```
program strana;       { *** 文中の文字、母音 *** }
var i : integer;
    str : string[255];
    vowels, set_ch : set of char;

begin
   vowels := ['A','E','I','O','U'];  {  母音字の集合  }
   set_ch := [];                     {  空集合  }
   writeln('文字列を打ち込んでください');
   readln(str);
   for i := 1 to length(str) do begin
     str[i] := upcase(str[i]);         {  小文字→大文字  }
     set_ch := set_ch + [str[i]];      {  入力文字の集合  }
   end;

   write('文字= ');
   for i := 0 to 255 do              {  集合要素の印刷  }
     if chr(i) in set_ch then write(chr(i));
   writeln;
   set_ch := set_ch * vowels;       {  母音字との積集合  }
   write('母音= ');
   for i := 0 to 255 do             {  集合要素の印刷  }
     if chr(i) in set_ch then write(chr(i));
```

```
    writeln;
end.
```

〈実 行 例〉

```
文字列を打ち込んでください
The marvellous thing is that it's painless.
文字=  '.AEGHILMNOPRSTUV
母音= AEIOU

文字列を打ち込んでください
The Salinas Valley is in Northern California.
文字=  .ACEFHILNORSTVY
母音= AEIO

文字列を打ち込んでください
It was the best of times, it was the worst of times,...
文字=  ,.ABEFHIMORSTW
母音= AEIO
```

このプログラムでは，小文字はすべて大文字に変換し，使用された文字の集合 str を作る．ここで和の演算を利用する．これを印刷し，次に母音字 vowels との積の演算により，使用された母音字を調べることができる．

5.3 レコード型

配列は，同じ型のデータが連続しているものであるが，それに対しレコード型はいろいろな型のデータをひとまとまりにして扱うために用いられる．

レコード型は次のように予約語 record と end で囲んで宣言する．

```
type date = record
                year : integer;
                month : (january, february, march, april, may, june,
                         july, august, september, october, november,
                         december);
                day : 1 .. 31;
            end;
```

要素となる型に制限はなく，使われた名前はフィールドとよばれる．レコード型の変数を宣言すると，それに含まれるフィールド分の記憶領域が確保される．また，レコード型を配列として扱うこともできる．

```
var today, tomorrow, yesterday : date;
    memorial : array [1...10] of date;
```

そして，レコード型変数の各要素を参照するのには，「変数名．フィールド名」というようにして表わされる名前を使う．また，変数名だけを指定してレコード全体のデータを代入することもで

きる.

```
today.year := 1986;
today.month := october;
today.day := 20;
memorial[1].year := 1955;
tomorrow := today;
memorial[3] := today
```

上記のようにdateというレコード型の宣言があり，さらに次のようにdateを用いたレコード型studentを宣言することもできる.

```
student=record
        name : string[20];
        sex : (male, female);
        birthday : date; { date はレコード型 }
      end;
```

このようなレコード型の入れ子では，

```
student.birthday.year := 1965;
```

というように，入れ子になっているフィールド名を順々に，

名前．フィールド名．…．フィールド名

というようにつないで参照する.

しかし，ある1つのレコード変数の要素を頻繁に参照するときに，常に変数名とフィールド名を並べるのは煩わしい．このような場合には，レコード変数を固定する働きを持つwith文を用いる．前述の例では

```
with today do begin
    year := 1986;
    month := october;
    day := 20;
end;
```

というようになる．with文のbegin endの間については指定したレコード変数名は省略され，フィールド名だけで変数への参照が行なえる.

[例題5.2] 個人データの作成

住所，性別，年齢，名前から構成される個人データを定義し，それらの入力と印刷を行なう.

プログラム 5.3

```
program pers_data;      { *** 住所録 *** }
type sextype = (male, female);
     person_range = 1..10;
     personal_data = record
                         sex :        sextype;
                         name :       string[20];
                         zip_code :   string[6];
                         address :    string[100];
                         age :        integer;
                     end;
var persons : array [person_range] of personal_data;
    i, j, data_max : integer;

procedure input_data(var person : personal_data);
begin
  with person do
  begin
    write('性別 0-男,1-女,2-終了:'); readln(j);
    if j = 0 then sex := male
    else if j = 1 then sex := female;
    if j <> 2 then begin
      write('名前    :'); readln(name);
      write('郵便番号:'); readln(zip_code);
      write('住所    :'); readln(address);
      write('年齢    :'); readln(age);
    end;
  end;
  writeln;
end;

procedure write_data(person : personal_data);
begin
  with person do
  begin
    write(name);
    case sex of
      male   : write(' 男性 ');
      female : write(' 女性 ');
    end;
    writeln(age,'才');
    write('〒',zip_code);
    writeln('  ',address);
    writeln;
  end;
end;

begin
  i := 0;
  repeat                    { データ入力 }
    i := i + 1;
    input_data(persons[i]);
  until
    j = 2;
  data_max := i - 1;    { データ出力 }
  for i := 1 to data_max do
    write_data(persons[i]);
end.
```

〈実 行 例〉

```
性別 0-男,1-女,2-終了:0
名前    :青木 英夫
郵便番号:300-11
住所    :茨城県牛久市小坂町××××－×
年齢    :20

性別 0-男,1-女,2-終了:1
名前    :森 裕子
郵便番号:213
住所    :神奈川県川崎市宮前区土橋×－×－××
年齢    :19

性別 0-男,1-女,2-終了:0
名前    :松田 淳一郎
郵便番号:352
住所    :埼玉県新座市栗原×－×－××
年齢    :21

性別 0-男,1-女,2-終了:1
名前    :堤 和子
郵便番号:166
住所    :東京都杉並区阿佐ヶ谷南×－×－××
年齢    :17

性別 0-男,1-女,2-終了:2

青木 英夫 男性 20才
〒300-11  茨城県牛久市小坂町××××－×

森 裕子 女性 19才
〒213  神奈川県川崎市宮前区土橋×－×－××

松田 淳一郎 男性 21才
〒352  埼玉県新座市栗原×－×－××

堤 和子 女性 17才
〒166  東京都杉並区阿佐ヶ谷南×－×－××
```

個人データは，名前や住所，性別，年齢から構成されるレコード型の配列である．入力を行なう手続き input_data，印刷を行なう手続き write_data とは1つのレコードを引数として渡している．

レコード型は，可変部も扱えるようになっていて，レコード型は何種類かの可変要素から構成されるものとして指定することができる．

```
type position = (pitcher,catcher,first,second,third,short,
                 left,center,right);
     playrec = record
                 name : string[20];
                 number : integer;
                 case pos : position of
                   pitcher : (win, lost, save : integer;
```

```
                    bougyoritsu : real);
            catcher..right : (daritsu : real;
                              daten,homerun : integer)
          end
        end;

var players : array[1..100] of playrec;
```

つまり，pos(これをタグフィールドという)の値がpitcherかそれ以外かによってレコードの内容が異なる．このような宣言をされた変数を扱う手続きでは，

```
procedure write_player(player : playrec);
begin
  with player do begin
    write(name);
    write(number : 5);
    case pos of
      pitcher : writeln(win : 3,'勝',lost : 3,'敗',save : 3,'セーブ',
                        '防御率',bougyoritsu : 5 : 2);
      catcher..right : writeln('打率',daritsu : 5 : 3,'打点',daten : 3,
                        'ホームラン',homerun : 3)
    end
  end
end;
```

型playrecが今問題としているものであり，この中にはタグによらない共通の要素(name, number)と可変部があり，可変部はposの値によって，(win, lost, save, bougyoritsu)と(daritsu, daten, homerun)とに分かれる．

タグの型は順序型に限られており，型名でしか指定できない．また，その型に属する定数はすべて，どれかの選択肢の前に現われていなければならない．なお，大きな順序型(整数型など)をタグの値とすることはできない．

5.4 動的データ構造

実際のデータ構造では，ポインタ型やレコード型が複雑に組み合わされていることが多い．

```
type linear_list = ^list_record;
     list_record = record
                     value : integer;
                     next : linear_list
```

```
                end;
```

上の宣言を見てみよう．list_record という型は，レコード型でフィールドとして，整数型の value と linear_list 型の next をもっている．ところが linear_list 型というのは，list_record 型へのポインタである．すなわちこの list_record 型は，自分自身へのポインタを含んでいることになっている．

このように，自分自身へのポインタを直接的あるいは間接的に含む型のことを，再帰的な型と呼ぶ．型の宣言については，ポインタで指される型に限って，後で定義する型名を先に使ってもよいことになっている．上の例では，linear_list 型の宣言に使った list_record がその例である．

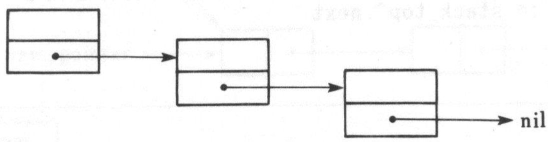


図 5.1

図 5.1 が示しているように，再帰的データは無限に続くことができる．それを終わらせる役目を持つのが nil である．

[例題 5.3]　スタックのシミュレーション

データ列の後に入れたものから順に取り出すスタックの動きを確かめる．

プログラム 5.4

```
program push_pop;        { *** 数値のプッシュとポップ *** }
type linear_list = ^list_record;
     list_record = record
                     value : integer;
                     next : linear_list
                   end;
var stack_top : linear_list;
    int, i : integer;

procedure push(var p : linear_list; i : integer);
var new_p : linear_list;
begin
  new(new_p);               { 新たな動的変数 }
  new_p^.value := i;
  new_p^.next := p;         { 一つ前のデータを指す }
  p := new_p                { 最後の変数を指すポインタを返す }
end;

function pop(var p : linear_list) : integer;
begin
  pop := p^.value;          { 引数の指していた値が関数値 }
  p := p^.next              { ポインタ値を一つ前のものにもどす }
end;

begin
```

```
    stack_top := nil;
    repeat
      write('push:1,pop:2,end:3 ? ');readln(i);
      case i of
        1 : begin
              write('push--数値 ?'); readln(int);
              push(stack_top,int)
            end;
        2 : writeln('pop --数値 = ',pop(stack_top))
      end;
    until
      not (i in [1,2]);
                                          { スタックの全要素の印刷 }
    write('All Pop = ');
    while stack_top <> nil do begin    { nil になるまで繰り返す }
      write(stack_top^.value : 5);     { pop の操作と同じ }
      stack_top := stack_top^.next
    end;
    writeln
  end.
```

〈実 行 例〉

```
push:1,pop:2,end:3 ? 1
push--数値 ? 12
push:1,pop:2,end:3 ? 1
push--数値 ? 34
push:1,pop:2,end:3 ? 1
push--数値 ? 56
push:1,pop:2,end:3 ? 1
push--数値 ? 78
push:1,pop:2,end:3 ? 2
pop --数値 = 78
push:1,pop:2,end:3 ? 2
pop --数値 = 56
push:1,pop:2,end:3 ? 1
push--数値 ? 1234
push:1,pop:2,end:3 ? 1
push--数値 ? 5678
push:1,pop:2,end:3 ? 3
All Pop =  5678 1234   34   12
```

このプログラムはスタックにデータを代入する手続き push，スタックからデータを取り出す関数 pop，メインプログラムから成る．push，pop は図 5.2 のようにポインタを利用してスタックにデータを付けたしたり，解放したりする．メインプログラムでは push や pop の呼び出しを行ない，最後にスタックにある全部のデータを印刷する．

このプログラムで用いたようなデータ構造を，線形リストという．次の例は線形リストを使ったソーティングのプログラムで，この方法をリスト挿入ソートという．

[例題 5.4] 線形リストによるソーティング

線形リストを利用してデータを小さい順に並べるソーティングを行なう．

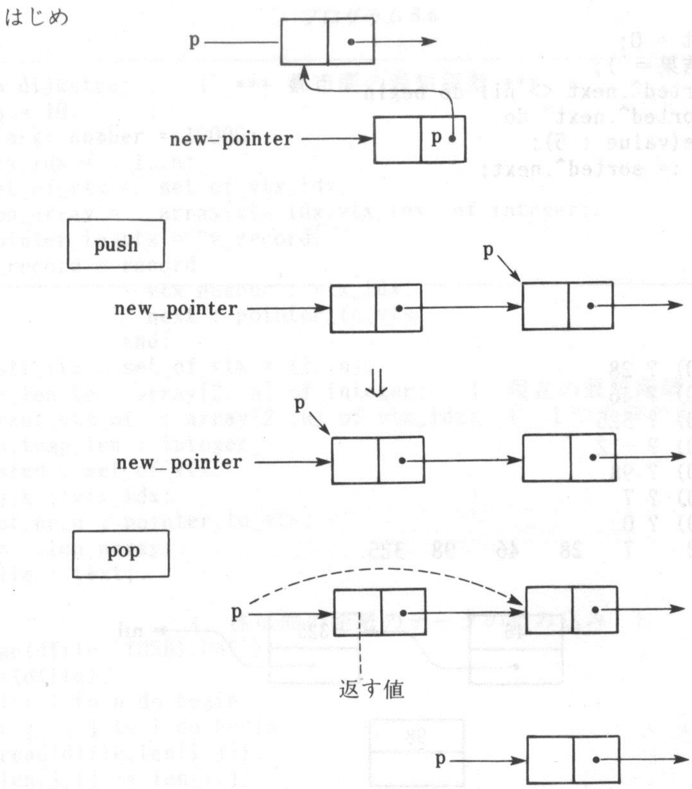


図 5.2

プログラム 5.5

```
program sorting_list;  { *** 線形リストによるソーティング *** }
type linear_list = ^list_record;
     list_record = record
                     value : integer;
                     next : linear_list;
                   end;
var sorted, compared, temp : linear_list;
    int : integer;

begin
  new(sorted);
  sorted^.next := nil;  { sorted^.next がリストの先頭 }
  repeat
    write('数値 (END=0) ? '); readln(int);
    if int <> 0 then begin
      compared := sorted;
                      { 値が int 以上になるまでリストを手繰る }
      while (compared^.next <> nil) and
            (compared^.next^.value < int) do
        compared := compared^.next;
      new(temp);
      temp^.value := int;
      temp^.next := compared^.next;
      compared^.next := temp;
```

```
      end
    until int = 0;
    write('結果=');
    while sorted^.next <> nil do begin
      with sorted^.next^ do
        write(value : 5);
      sorted := sorted^.next;
    end;
    writeln;
  end.
```

〈実 行 例〉

```
数値 (END=0) ? 28
数値 (END=0) ? 46
数値 (END=0) ? 325
数値 (END=0) ? -12
数値 (END=0) ? 98
数値 (END=0) ? 7
数値 (END=0) ? 0
結果=   -12    7   28   46   98  325
```

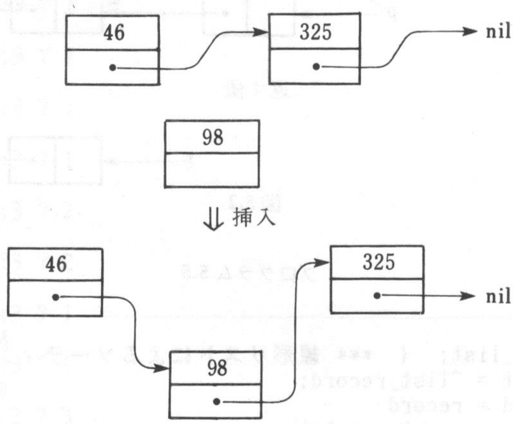


図 5.3

このプログラムでは，図5.3のように数値の大小比較を行ないながら，線形リストを生成していく．数値として0が入力された場合，これらのリストのデータを順に印刷して終了する．

最後の例は，集合型と線形リスト構造をうまく利用した例で，やや複雑な例ではあるが，n個の点(都市)がありそれぞれ相互の距離が与えられているとき，どういう経路を通るのが最短路であるかを探すプログラムである．

[例題 5.5] 都市間の最短距離

n個の点と相互間の距離が与えられているとき，1番目の都市から$i(2 \leq i \leq n)$番目の都市へ行く最短路を探索する．

プログラム **5.6**

```
program dijkstra;      { *** 都市間の最短距離 *** }
const n = 10;
      large_number = 10000;
type vtx_idx =  1..n;
     set_of_vtx =  set of vtx_idx;
     len_array =  array[vtx_idx,vtx_idx] of integer;
     pointer_to_vtx = ^v_record;
     v_record = record
                  vtx_number : vtx_idx;
                  next : pointer_to_vtx
                end;
const all_vtx : set_of_vtx = [1..n];
var min_len_to : array[2..n] of integer;    { 現在の最短距離 }
    parent_vtx_of : array[2..n] of vtx_idx;  {  1つ手前の点 }
    min,temp_len : integer;
    tested : set_of_vtx;
    i,j,k : vtx_idx;
    root,pp,q : pointer_to_vtx;
    len : len_array;
    dfile : text;

begin                        { 都市間の距離のデータの読み込み }
  assign(dfile,'TOSHI.DAT');
  reset(dfile);
  for i:= 1 to n do begin
    for j := 1 to i do begin
      read(dfile,len[i,j]);
      len[j,i] := len[i,j]
    end;
    readln(dfile);
  end;
  close(dfile);
  tested := [1];             { tested は調べた都市の number }
  for i := 1 to n do begin
    min_len_to[i] := len[1,i];  {  初期値は1からの距離 }
    parent_vtx_of[i] := 1;        { parent は1 }
  end;
  while tested <> all_vtx do begin  {  全部調べるまで続く }
    min := large_number;
    for i := 2 to n do
        { tested になく min_len が一番小さなものを探す }
      if (i in (all_vtx - tested)) and
         (min_len_to[i] < min) then begin
        k := i;
        min := min_len_to[i];
      end;
    tested := tested + [k]; {  その都市を tested に加える }
    for i := 2 to n do
      if (i in (all_vtx - tested)) then begin
        temp_len := min_len_to[k] + len[k,i];
          { k 経由が短いときは parent を k にして  }
                   { k 経由の距離を最短距離とする  }
        if min_len_to[i] > temp_len then begin
          min_len_to[i] := temp_len;
          parent_vtx_of[i] := k;
        end;
      end;
  end;
```

```
  for j := 2 to n do begin
    write('1から ',j,' への最短路');
    new(pp);
    pp^.vtx_number := j;
    pp^.next := nil;
    root := pp;
    i := j;
    repeat
      i := parent_vtx_of[i];    { 1にぶつかるまで親を手繰っていく }
      new(pp);
      pp^.vtx_number := i;
      pp^.next := root;
      root := pp;
    until i = 1;
    pp := root;
    while pp <> nil do begin
      write(pp^.vtx_number : 3);
      pp := pp^.next;
    end;
    writeln;
    writeln('  最短距離は ',min_len_to[j]);
    writeln;
  end;
end.
```

〈TOSHI.DAT の内容〉

```
  0
142   0
 85  92   0
 62  96  30   0
 47  20  89  59   0
 38 120  36  83  34   0
113  57  35  51  99 126   0
113  29  83  38  75  94  53   0
116  35 107  57  78  86  80  25   0
 80  90 187 108  57  41 127  75  56   0
```

〈実 行 例〉

```
1から 2 への最短路  1  5  2
  最短距離は 67

1から 3 への最短路  1  6  3
  最短距離は 74

1から 4 への最短路  1  4
  最短距離は 62

1から 5 への最短路  1  5
  最短距離は 47

1から 6 への最短路  1  6
  最短距離は 38

1から 7 への最短路  1  6  3  7
  最短距離は 109
```

```
1から 8 への最短路  1  5  2  8
  最短距離は 96

1から 9 への最短路  1  5  2  9
  最短距離は 102

1から 10 への最短路  1  6 10
  最短距離は 79
```

一例として1から2への最短路を考える．1から各点への距離は次のようになる．

	2	3	4	5	6	7	8	9	10
1からの距離	142	85	62	47	38	113	113	116	80
親	1	1	1	1	1	1	1	1	1

1からは6が一番近く，1から6に行く場合これ以上近道がないことは明らかである．したがって6は集合testedに入れられる．

次に6を経由した方が近くなる点を探すと，3は74で85より小さく，10は6を通ると79となり現在の80より短くなる．そこで，3と10の親を6につけかえる．

2	3	4	5	⑥	7	8	9	10
142	74	62	47	38	113	113	116	79
1	6	1	1	1	1	1	1	6

6の次に選ばれるのは5であり，これより1-2間よりさらに短い道がみつかる．

2	3	4	⑤	⑥	7	8	9	10
67	74	62	47	38	113	113	116	79
5	6	1	1	1	1	1	1	6

このようにして，1からi ($i \leq 2 \leq 10$)番目の点への最短路を索していく．

```
for j := 3 to n do begin
  write('1から', [illegible] 'への最短路');
  new(pp);
  pp^.vtx_number := j;
  pp^.next := nil;
  root := pp;
  i := j;
  repeat
    [illegible]
    new(pp);
    pp^.vtx_number := i;
    pp^.next := root;
    root := pp
  until i = 1;
  pp := root;
  [illegible]
    write(pp^.vtx_number[illegible]);
    pp := pp^.next
  end;
  writeln([illegible] min_len[j][illegible]);
  writeln;
 end;
end.
```

0									
142	0								
[illegible]	92	0							
62	96	30	0						
[illegible]	20	89	52	0					
[illegible]	120	[illegible]	[illegible]	[illegible]	[illegible]				
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]			
[illegible]	[illegible]	[illegible]	[illegible]	73	[illegible]	[illegible]	0		
[illegible]	[illegible]	[illegible]	[illegible]	78	86	[illegible]	[illegible]	0	
80	90	18	[illegible]	57	[illegible]	[illegible]	[illegible]	[illegible]	0

実行例

1から 2 への最短路 1 5 2
最短距離は [illegible]

1から 3 への最短路 1 [illegible]
最短距離は 74

1から 4 への最短路 [illegible]
最短距離は 62

1から 5 への最短路 1 [illegible]
最短距離は 47

1から 6 への最短路 1 [illegible]
最短距離は [illegible]

1から 7 への最短路 1 [illegible]
最短距離は 109

第6章 ユニット

TURBO Pascal ver. 5では，数値計算や文字・ファイルの取り扱いのために数多くの手続き・関数が標準で用意されている．これらについては第3章で取り上げた．これ以外にグラフィックスやオペレーティングシステムに関する拡張された機能も用意されているが，それらはユニットと呼ばれるまとまった機能に分類されている．

ユニットには，Crt，Dos，Graph，Overlay，Turbo3などがあり，例えばグラフィックを使うためにはGraphユニットを結合してプログラムを作成する．これらの中でもよく用いられるCrt，Dos，Graphの3つのユニットについて説明する．

6.1　Crtユニット

Crtユニットには，ディスプレイやキーボードに関するいろいろな手続き・関数がある．例えば，文字に色をつけたり，画面上の任意の場所に文字を表示したり，キーボードのキーを打った直後にそのキーの文字を入力したりすることができる．代表的なものには次のような手続き・関数がある．

ClrScr	画面を消去し，カーソルを上左端に置く．
Delay (ms)	msミリ秒の遅延を行う．DelayFactorの値で調節する．
GotoXY (x, y)	カーソルを(x, y)に移動する．
TextColor (Color)	文字の色属性を設定する．
x：= KeyPressed	キーが押されていればtrueを返す．
c：= ReadKey	キーボードから1文字読み込む．画面上には表示されない．入力がないときは，入力されるまで待つ．
i：= WhereX	カーソルのx座標を返す．
i：= WhereY	カーソルのy座標を返す．

この中でReadKeyはKeyPressedと一緒にして使うと，押したキーを読み取ることができる．

[例題 6.1] キーボード練習

A～Z の文字を画面に表示し，プレイヤーに同じキーを打たせる．同じ文字であれば点数が加算され，一定時間内にキー入力がないと，次に進む．

プログラム 6.1

```
{  *** キーボード練習 ***  }
program key_touch;
uses Crt;
label next;
var i, j, k, s : integer;
    ch, ch2 : char;

begin
  write('上級  1←→5  初級 ?');
  readln(j);
  j:=j*1000;
  randomize;
  clrscr;
  gotoxy(6,10);
  write('*** キーボード練習(30回) ***');
  s := 0;
  for i := 1 to 30 do begin
    gotoxy(10,12); write(i:2,' 回目');
    ch2 := chr($41 + random(26));   { ch2='A'～'Z' }
    gotoxy(16,14); write(ch2,' - ');
    for k := 1 to j do begin
      if keypressed then begin  {  キー押下  }
        ch := upcase(ReadKey);
        write(ch);
        if ch = ch2 then begin  {  一致  }
          s := s + 1;
           gotoxy(22,12); write(s:2,' 点')
        end;
        delay(300);
        gotoxy(16,14); write(' ':5);
        goto next
      end
    end;
next:
  end;
  gotoxy(16,14); writeln('終了しました.')
end.
```

画面上には，横80字(日本語で40字)縦24行の文字が表示できる．文字列の表示は，普通はスクロールされて消えていってしまうが，画面上に座標を指定して，特定の場所に文字列を表示することができる．

gotoxy(x, y);　　x, y は整数型

という手続きを呼び出して，この次に write 文で文字列を表示すると xy で示した座標から文字を表示する．

6.2 Dos ユニット

Dos ユニットにはオペレーティングシステムやファイルを取り扱う数多くの機能がある．例えば現在の日付，時刻を取得したり，ディレクトリにあるファイルの名前，属性などの情報を得ることができる．これにより計算時間を調べたりすることができる．Dos ユニットの手続き・関数は付録参照のこと．

[例題 6.2]　現在の日付と時刻

現在の日付，時刻を得て，それを表示する．

プログラム 6.2

```
{ *** 現在の日付と時刻 *** }
program date_time;
uses Dos;
const Week:string[14] = '日月火水木金土';
var Year,Month,Day,DofWeek,
    Hour,Minute,Second,Sec100:word;
begin;
  GetDate(Year,Month,Day,DofWeek);
  GetTime(Hour,Minute,Second,Sec100);
  writeln(Year,'年',Month,'月',Day,'日(',
    copy(Week,DofWeek*2+1,2),') ',
    Hour,'時',Minute,'分',Second,'秒')
end.
```

このプログラムはコンピュータに内蔵されている時計の日付・時刻を読んで，それを表示するものである．

次の例は MS-DOS の dir コマンドと同じ働きをするプログラムである．

[例題 6.3] ファイル名の一覧

現在のディレクトリのファイル名一覧を表示する.

プログラム 6.3

```
{  *** ファイル名の一覧 ***  }
program file_names;
uses Dos;
var Dir:SearchRec;
begin
  FindFirst('*.*',Archive,Dir);
  while DosError=0 do begin
    Writeln(Dir.Name,' ',Dir.Size);
    FindNext(Dir)
  end
end.
```

6.3 Graph ユニット

はじめに Graph ユニットを理解するために図形表示のしくみについて説明する.

(a) 図形表示のしくみ

パーソナルコンピュータでは,画面上に文字とは別にいろいろな図形を表示することができる.図のように,画面上の横 1120×縦 750 個(パソコンの機種によっては横 640×縦 400 個)の点に色を

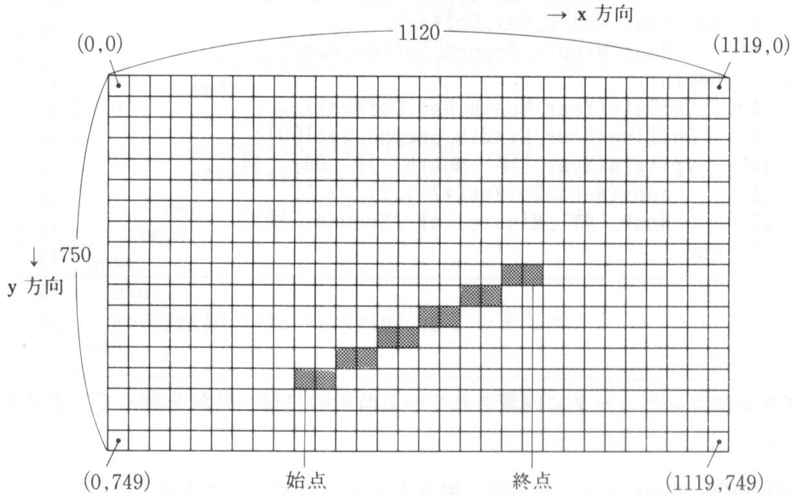


図 6.1 グラフィックスの座標

つけることによりいろいろな図形を表示するしくみである．また画面消去や線分の描画などの機能を備えており，たとえば図のように始点と終点の座標値を示せば，線分を描画することができる．これらの点(pixel ともいう)の座標は，横方向が0から1119まで，縦方向が0から749までで，左上の点が(0, 0)，右下の点が(1119, 749)になっている．

各pixelには，赤緑青(RGB)，輝度のON/OFFで2の4乗＝16通りの色を付けることができる．既定値は次のようになっているが変更もできる．

輝	緑	赤	青	色	色コード	輝	緑	赤	青	色	色コード
0	0	0	0	黒	0	1	0	0	0	灰	8
0	0	0	1	青(暗)	1	1	0	0	1	青	9
0	0	1	0	赤(暗)	2	1	0	1	0	赤	10
0	0	1	1	紫(暗)	3	1	0	1	1	紫	11
0	1	0	0	緑(暗)	4	1	1	0	0	緑	12
0	1	0	1	水(暗)	5	1	1	0	1	水	13
0	1	1	0	黄(暗)	6	1	1	1	0	黄	14
0	1	1	1	白(暗)	7	1	1	1	1	白	15

(b)　基本機能

Graphユニットには，グラフィックスの機能が多数用意されていて，uses graph；という宣言で使用できる．Graphユニットの代表的な機能をごく一部紹介する．

Detect	装置の検査(整数型関数)
InitGraph(gd, gm, ")	初期化，画面消去
CloseGraph	終了
ClearDevice	画面消去
Line(x1, y1, x2, y2)	線分の描画始点(x1, y1)と終点(x2, y2)を結ぶ
MoveTo(x, y)	点を(x, y)に移動する．線は引かない
LineTo(x, y)	現在の点から点(x, y)まで線を引く
SetColor(c)	色の設定，cは色コード
MoveRel(dx, dy)	相対移動
LineRel(dx, dy)	相対描画
SetRGBpalette(i, R, G, B)	i番目のパレットに色を設定する
PutPixel(x, y, c)	点(x, y)に色cの点を打つ
GetMaxX	x方向の最大値(関数)

GetMaxY	y方向の最大値(関数)

グラフィックスを利用する場合は，uses graph；の宣言以外に，はじめにDetect, InitGraphの初期化機能を呼び出さないといけない．最後にCloseGraphを呼び出し，その間必要に応じて，線分の描画や色の設定などの機能を利用することになる．

例えば，画面に三角形を表示するプログラムは次のようになる．色コードの規定値は15(白)である．

```
 1  program tri(input,output);
 2  uses graph;
 3  var gd,gm:integer;
 4  begin
 5    gd:=Detect;           { 初期化 }
 6    InitGraph(gd,gm,'');
 7    Line(300,500,800,500);
 8    Line(800,500,600,200);
 9    Line(600,200,300,500);
10    readln;     {⏎キーを押して画面が消える}
11    CloseGraph
12  end.
```

プログラム6行目Init Graphの3番目の引数はグラフィックスの実行に必要なBGIファイルの存在するディレクトリを指示する*．カレントディレクトリにあるときはこのように空文字列にする．

なお，最後のCloseGraphを使うとグラフィックの画面が消去されてしまうので，⏎キーを押すまで画面を残しておくためにreadlnが入っている．

(c) 直線図形

グラフィックスの基礎は直線を画面に表示することである．座標計算の手順がわかれば座標値を順に線分で結び，図形を表示することができる．

[例題6.4] ランダムウォーク

画面中央の点から方向を乱数で決めて短い線を引き，それを繰り返す．

* 情報教育棟のFMRでは3番目の引数を 'e：¥pascal' とする．ここを '' にするときはファイル e：¥pascal¥fmrh. bgi をカレントディレクトリにコピーしてから実行する．

プログラム 6.4

```
{  *** ランダムウォーク ***  }
program random_walk(output);
uses Crt,Graph;
var gd,gm,x,y,i : integer; t:real;

begin
  gd := Detect;
  InitGraph(gd,gm,'');
  ClearDevice;
  Randomize;
  x:=GetMaxX div 2;
  y:=GetMaxY div 2;
  moveTo(x,y);          {  画面中央の点  }
  i:=0;
  repeat
    i:=i+1;
    setColor(i div 100 mod 7+9);
    t:=random*2*pi;
    x:=x+round(5*cos(t));
    y:=y+round(5*sin(t));
    LineTo(x,y);
    Delay(50)
  until KeyPressed;
  readln;
  CloseGraph
end.
```

この例題では，graph ユニット以外に crt ユニットを使っている．crt ユニットはグラフィックス以外の画面制御などのユーティリティが含まれている．

delay(i)　は時間稼ぎ(表示を遅くするため)をする手続きである．

KeyPressed　は何かキーが押されれば true を返す関数である．ここでは何かキー(例えばスペースキー)を押すと繰り返しを中止する．

画面中央の点は x, y の最大値を 2 で割って求める．random は引数がないと，0 以上 1 未満の一様乱数を得ることができる．

次に幾何学的な図形の例として，ダイヤモンドパターンと呼ばれる図形を表示する．表示方法は始めに正多角形の頂点座標値を計算し，次にこれらの頂点間を直線で結ぶ．

[例題 6.5]　ダイヤモンドパターン

正多角形のすべての頂点を直線で結び，ダイヤモンドパターンと呼ばれる図形の表示を行う．

プログラム 6.5

```
1  {  *** ダイヤモンドパターン *** }
2  program diamond;
3  uses Graph;
4  var gd, gm, i, j, np : integer;
5      x, y : array[1..100] of integer; {  座標値  }
6      rs, ang : real;    {  半径、角度  }
7  
8  begin
9    gd := Detect;
10   InitGraph(gd,gm,'');
11   write('頂点数  ? '); readln(np);
12   rs := GetMaxY/2.2;
13   for i := 1 to np do begin    {  正多角形の頂点  }
14     ang := 2*pi*(i-1)/np;
15     x[i] := round(rs*cos(ang) + GetMaxX div 2);
16     y[i] := round(rs*sin(ang) + GetMaxy div 2);
17   end;
18   for i := 1 to np - 1  do    {  頂点間の線分描画  }
19     for j := i + 1 to np do
20       line(x[i],y[i],x[j],y[j]);
21   readln;
22   CloseGraph
23 end.
```

プログラムの 12〜17 行が正多角形の頂点座標値の計算部分である．円周上を n 等分した点の座標を計算する．円の半径は，画面 y 方向の最大値の半分よりわずかに小さい値に設定する (12 行)．円の中心は画面の中心に等しくする．計算された座標値は配列に記憶しておく．

後半の 18〜20 行が表示部分である．この部分は n 個の中から異なる 2 個をすべて選び出す繰り

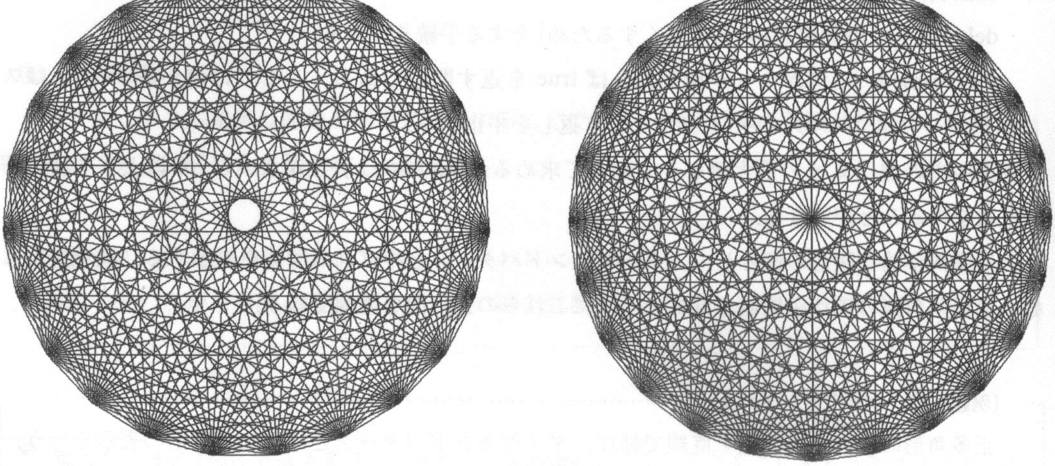


図 6.2

返し処理である．

表示例を図 6.2 に示す．左が頂点数 23，右が 24 の表示例である．頂点数が奇数のときは図形の中央に空白部分ができ，偶数のときは放射状に線が通過する．このプログラムはプログラムの変更なしに，パソコンの機種に依存しないで図形を表示することができる．

(d)　曲線図形

曲線図形の作図は，基本的には曲線を微小区間に分割し，各区間を直線とみなして線を描画することにより作図できる．どれくらいの長さに分割するかは，曲線の曲率や画面の解像度に依存する(図 6.3 参照)．

[例題 6.6]　極座標図形

$r = a\sin(p\,\theta/q)$ という極座標図形を表示する．

プログラム 6.6

```
{  *** 極座標図形 ***  }
program rose(input,output);
uses Graph;

var gd,gm,p,q,np,i,x,y : integer;
    th,rs : real;

begin
  write('ｐ，ｑの値　？');
  readln(p,q);
  np := 400 * q;          {  繰返し回数  }
  if (p mod 2 = 1) and
     (q mod 2 = 1) then np := np div 2;
  gd := Detect;
  InitGraph(gd,gm,'');
  ClearDevice;
  for i := 0 to np do begin
    th := 2.0 * PI * i / 400;    {  角度  }
    rs := GetMaxy/2.2 * sin(p * th / q);
    x := round( rs * sin(th) + GetMaxX/2);
    y := round(-rs * cos(th) + GetMaxY/2);
    if i > 0  then lineto(x,y)   {  線素の描画  }
              else moveto(x,y)
  end;
  readln;
  CloseGraph
end.
```

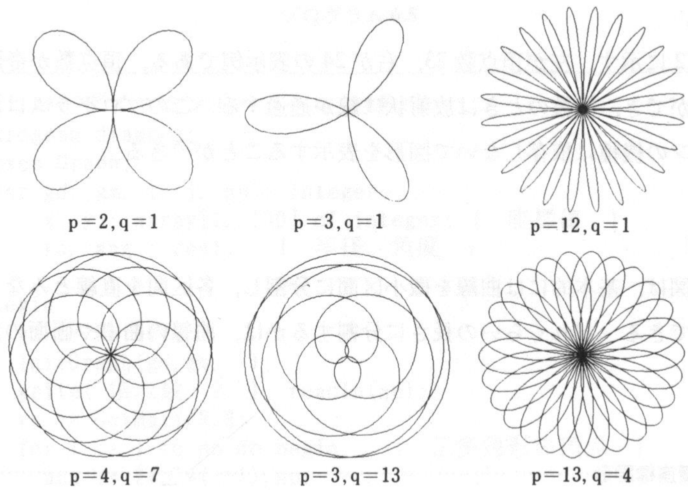


図 6.3

(e) 再帰的図形

再帰的な図形の例として，直線を再帰的に2つに分枝させる樹形図と，多角形の辺を再帰的に変形させていくコッホ曲線の生成と表示を行う．

[例題 6.7] 樹形図の作成

線分を次々に2股に分枝させて樹形図を作成する．

プログラム 6.7

```
{  *** 再帰用法による樹形図 ***  }
program tree_disp;
uses Graph;
var gd, gm :integer;

procedure tree(n,x,y:integer;ang,len:real);
var x1,y1 : integer;
begin
  if n>0 then begin
    x1 := round(x+len*cos(ang*pi/180));
    y1 := round(y-len*sin(ang*pi/180));
    line(x,y,x1,y1);
    ang := ang*(random*0.1+0.95);
    len := len*(random*0.1+0.95);
    tree(n-1,x1,y1,ang-15,len/1.3);   {  2分技  }
    tree(n-1,x1,y1,ang+15,len/1.3);
  end;
end;

begin
```

```
21    gd := Detect;
22    InitGraph(gd,gm,'');
23    randomize;
24    tree(10,GetMaxX div 2,GetMaxY,90,GetMaxY div 4);
25    readln;
26    CloseGraph
27  end.
```

プログラムの中心部は6〜18行の手続きである．引数で示された始点(x, y)と線分の方向(ang)，線分の長さ(len)にしたがって，始点から1本の線分を描画する(10〜12行)．次に自然に近い樹形図を作るため，枝の角度と長さを乱数で変化をつける．そして2又に分枝する2本の線分を描く(15，16行)．角度は左右に15°ずつずらし，枝の長さを少し短くする．

メインプログラムでは，グラフィックスと乱数の初期化を行って，上の手続きを呼びだすだけである．始点の座標，枝の長さの初期値はパソコンの機種に依存しないようにした．角度の初期値は90°である．

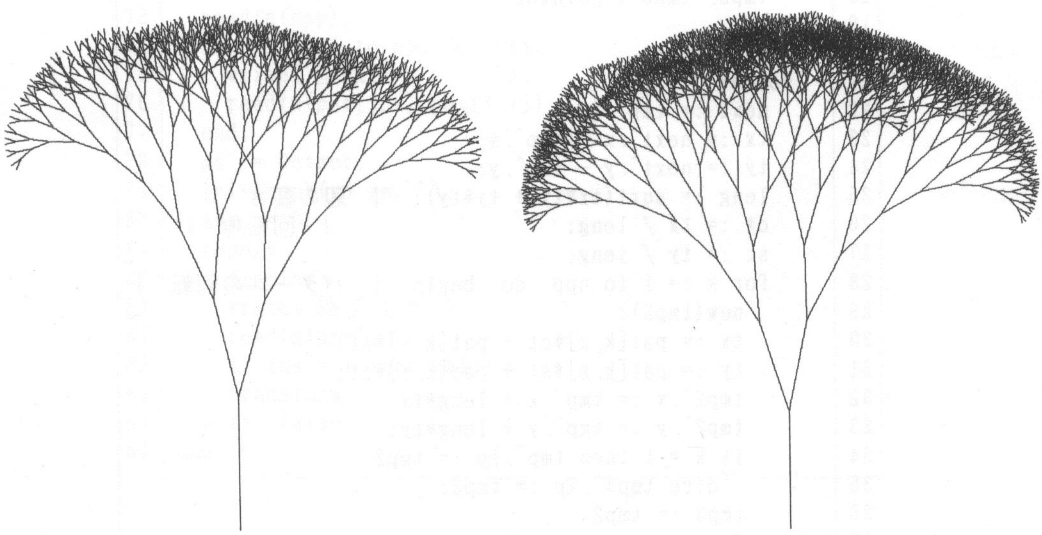


図 6.4

再帰で注意するのが，再帰のレベルの深さである．最初の引数nがこれを制御する変数であり，初期値が10のときは根元から葉の先まで10段階分枝する．出力図(図6.4)の左側はn = 10，右側はn = 12の例である．

再帰的な図形の例として，多角形の辺を再帰的に変形させていくコッホ曲線の生成と表示を行う．

[例題 6.8]　コッホ曲線

動的データ構造を使ってコッホ曲線の描画を行う．

プログラム 6.8

```
{ *** コッホ曲線の描画 *** }
program koch(input,output);
uses Graph;

type pointer = ^cell;
     cell = record
       ip : pointer;
       x : real;
       y : real;
     end;
var root, tmp, next : pointer;
    np, npp : integer;
    pat : array[1..18,(x,y)] of real;

procedure transform;     { 自己相似変形 }
var k : integer;
    leng, ct, st, tx, ty : real;
    tmp2, tmp3 : pointer;
begin
  tmp := root;
  repeat
    next := tmp^.ip;
    tx := next^.x - tmp^.x;
    ty := next^.y - tmp^.y;
    leng := sqrt(tx*tx + ty*ty);  { 辺の長さ }
    ct := tx / leng;                  { 回転角 }
    st := ty / leng;
    for k := 1 to npp  do  begin  { パターンの回転 }
      new(tmp2);
      tx := pat[k,x]*ct - pat[k,y]*st;
      ty := pat[k,x]*st + pat[k,y]*ct;
      tmp2^.x := tmp^.x + leng*tx;
      tmp2^.y := tmp^.y + leng*ty;
      if k = 1 then tmp^.ip := tmp2
        else tmp3^.ip := tmp2;
      tmp3 := tmp2
    end;
    tmp2^.ip := next;
    tmp := next
  until next^.ip = nil
end;

procedure display;          { 多角形の表示 }
var x, y : integer;
begin
  ClearDevice;
  tmp := root;
  repeat
    x := round( tmp^.x + GetMaxX/2);
    y := round(-tmp^.y + GetMaxY/2);
    if tmp <> root then lineto(x,y)
```

```
                    else moveto(x,y);
    tmp := tmp^.ip
  until tmp = nil
end;

var gd, gm, i, ans : integer;  {  メインプログラム  }
begin
  write('頂点数 ? ');    {  多角形の入力  }
  readln(np);
  new(root);
  tmp := root;
  for i:=1 to np do begin
    write('座標値(x y) ? ');
    readln(tmp^.x,tmp^.y);
    new(tmp^.ip);
    tmp := tmp^.ip
  end;
  tmp^.x := root^.x;
  tmp^.y := root^.y;
  tmp^.ip := nil;
  write('頂点数 ? ' );    {  変形パターンの入力  }
  readln(npp);
  for  i := 1 to npp do begin
    write('座標値(x y) ? ');
    readln(pat[i,x],pat[i,y])
  end;
  gd := Detect;
  InitGraph(gd,gm,'');
  ans := 0;
  repeat
    display;
    write('終了:1 ? ');
    readln(ans);
    if ans = 1 then halt;
    transform
  until false
end.
```

図6.5のように図形を構成する線分をある規則により複数個の連続した線分に分け，これらの各線分に対して同じ規則を次々に適用していくことにより得られる図形がコッホ曲線である．線分を一つなぎの複数個の線分に分割し，これらの線分に対して同じ変形規則を適用するプログラムを作成する．これを図6.6のように座標データをレコード型のセルで表し，ポインタを付け変えることにより，図形を変形させてコッホ曲線の作成と表示を行なうのが，例題6.8である．

メインプログラムでは，最初の頂点列のデータと変形するためのデータを入力する．頂点のデータの座標系は横方向が−320から＋320，縦方向が−200から＋200になっている．変形パターンのデータは，図6.5(2)のように(0,0)から始まり(1,0)で終わる頂点列の中間の点の数とそれらの点の座標値である．正3角形に図6.5(2)のような規則を適用することにより得られる図形は，雪片曲線と呼ばれる(図6.7上)．

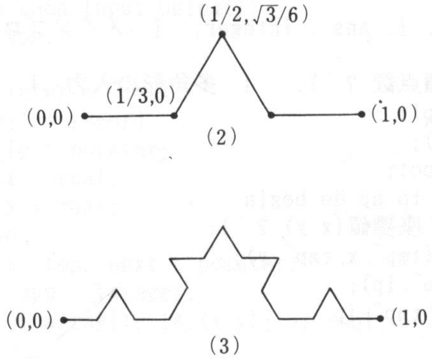


図 6.5　辺の変形

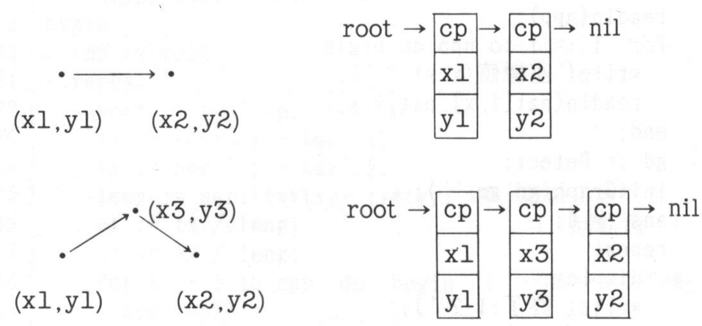


図 6.6　頂点の追加

このプログラムでは雪片曲線以外にも，入力データを変更することによりいろいろなコッホ曲線を描くことができる．図6.7中，下は正方形を変形させた例である．

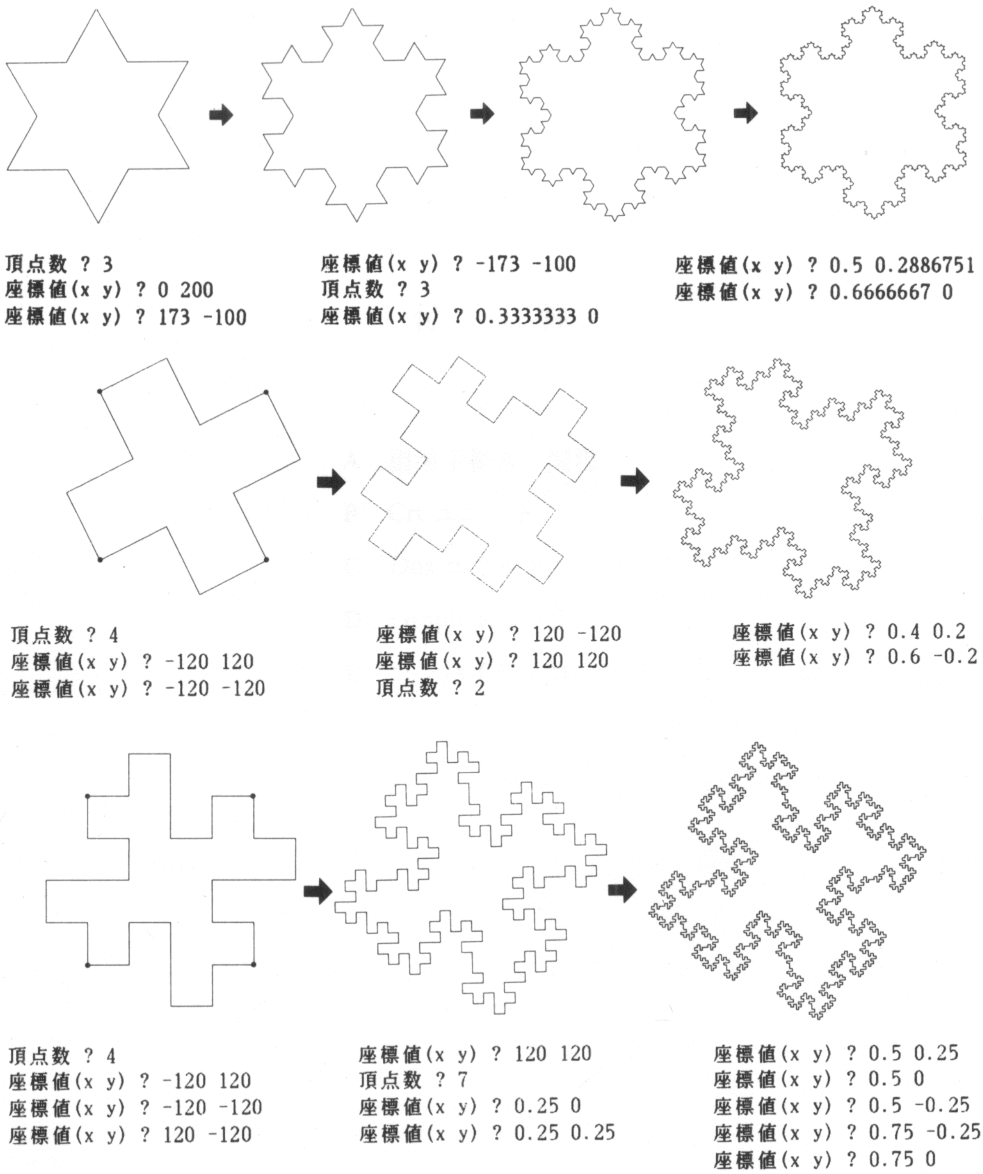


図 6.7　いろいろなコッホ曲線

付　　　録

付録A　標準手続き・関数

1. ファイル

・Assign (var f ; name ： string)　　ファイル変数 f をファイル名 name に結び付ける．f はあらゆるファイル型のファイル変数．name にはドライブ，ディレクトリ名も含めることができる(最大 79 文字)．name の文字数が 0 のとき標準入出力として定義される．

・ChDir (s：string)　　カレントディレクトリを s で示すドライブ，ディレクトリに変更．{$ I−} の場合，IOResult にエラーコードが返される．

・Close (var f)　　オープンされたファイルのクローズ．{$ I−} の場合，IOResult にエラーコードが返される．

・Erase (var f)　　f が示す外部ディスクファイルの消去．オープンしたファイルに用いてはならない．{$ I−} の場合，IOResult にエラーコードが返される．

・GetDir (d：byte ; var s：string)　　d で示されたドライブのカレントディレクトリを s に代入する (d = 0 がカレントドライブ，d = 1 がドライブ A，d = 2 がドライブ B……)．{$ I−} の場合，IOResult にエラーコードが返される．

・MkDir (s：string)　　s で示されたサブディレクトリを新規作成する．{$ I−} の場合，IOResult にエラーコードが返される．

・Rename (var f ; newname：string)　　f が示すディスクファイルの名前の変更．
{$ I−} の場合，IOResult にエラーコードが返される．

・Reset (var f)
Reset (var f：file ; recsize：word)　　既存のファイルをオープンする．f は Assign で定義されたファイル変数．f が型なしファイルの場合，recsize を指定することができる．これはデータ転送時のレコードサイズで，省略すると 128byte となる．f がすでにオープンされている場合，一度クローズしてから再度オープンする．また f がテキストファイルの場合読み込み専用となる．{$ I−} の場合，IOResult にエラーコードが返される．

・Rewrite (var f)
Rewrite (var f：file ; recsize：word)　　新しいファイルを作ってオープンする．f は Assign で定義されたファイル変数である．f が型なしファイルの場合，recsize を指定することができる．これはデータ転送時のレコードサイズで，省略すると 128byte となる．f がすでにオープンされている場合，一度削除してから再度新しいファイルをオープンする．また，f がテキストファイルの場合出力専用となる．{$ I−} の場合，IOResult にエラーコードが返される．

・RmDir (s：string)　　s で示された空のサブディレクトリを消去する．{$ I−} の場合，IOResult にエラーコードが返される．

・b：= Eof (var f)　　ファイルの終わりの状態を論理型で返す．f はファイル変数で，ファイル最後の要素を越えている場合，true を返す．テキスト型で標準入力 Input の場合，f を省略することができる．{$ I−} の場合，IOResult にエラーコードが返される．

・w：= IOResult　最後に行われた I/O 操作の状態を示す値を返す．結果は word 型．結果が 0 は正常だったことを意味する．なお，デフォルトは {$I+} なので，{$I−} にしておく必要がある．

2. テキストファイル

・Append (var f：text)　既存のファイルを追加モードでオープンし，ファイルポインタを最後に移す．{$I−} の場合，IOResult にエラーコードが返される．

・Flush (var f：text)　出力用にオープンされているセクタバッファをファイルに書き込んで空にする．{$I−} の場合，IOResult にエラーコードが返される．

・SetTextBuf (var f：text；var buf [；size：word])　I/O バッファをテキストファイルに割り当てる．デフォルトは 128byte なので，このバッファを大きくするときに使用する．Reset, Rewrite, Append の直後に呼び出す．size を省略すると，SizeOf (buf) となる (自動的に大きさを計算する)．

・b：= Eoln (f)　行の終わりの判定

・b：= SeekEof (f)　ブランク，タブをスキップしたファイルの終わりの判定

・b：= SeekEoln (f)　ブランク，タブをスキップした行の終わりの判定

3. 型なしファイル

・BlockRead (var f：file；var buf；count：word [；var result：word)　ファイル f から buf の先頭へ count レコードを転送する．result は実際に転送されたレコード数．レコードサイズは reset で指定する．

・BlockWrite (var f：file；var buf；count：word [；var result：word)　buf の先頭から count レコードをファイル f に転送する．result は実際に転送されたレコード数．

・L：= filepos (var f)　ポインタが現在いる位置

・L：= filesize (var f)　ディスクファイルのサイズ (要素数)．

・Seek (var f；n：longint)　ファイルポインタをファイルの n 番目の要素に移動する．

・Truncate (var f)　現在のファイルポインタの先の部分を削除する．

4. フロー制御

・Exit　現在のブロックから脱出する．

・Halt [(exitcode：word)]　プログラムの実行を停止し，オペレーティングシステムに戻る．引数がないと，0 を指定したのと同じになる．

・RunError [(ErrorCode：word)]　プログラムの実行を止め，実行時エラーを出す．

5. 算術関数・乱数

・Randomize　乱数発生の初期化をおこなう．乱数値はシステムクロックから得る．

・y：= Abs (x)　引数の絶対値を返す．引数は整数型か実数型．結果の型は引数の型に等しい．

・y：= ArcTan (x：real)　逆正接関数 (アークタンジェント)．結果はラジアン．

・y：= Cos (x：real)　余弦関数 (コサイン)．

・y：= Exp (x：real)　指数関数．e (自然対数の底) の x 乗に等しい．

・y：= Frac (x：real)　引数の小数部を返す．引数が負の場合は結果も負となる．

・y：= Int(x：real)　引数の整数部を返す．結果も実数型である．引数が負の場合は結果も負となる．

・y：= Ln(x：real)　自然対数を返す．

・y：= Pi　円周率πの値を返す．

・y：= Random[(range：word)]　rangeが指定されていない場合，0.0以上1.0未満の一様乱数を作る．指定されているときは，0以上range未満の乱数値(word型)が返される．乱数の初期化はrandomizeを呼ぶか，変数randseed(longint型)に値を設定する．

・y：= Sin(x：real)　正弦関数(サイン)．

・y：= Sqr(x)　引数の2乗を返す．xは整数型または実数型の式で，結果の型は引数の型に等しい．

・y：= Sqrt(x：real)　引数の平方根を返す．負の場合は実行時エラーになる．

6.　順序型・型変換関数

・Dec(var x[；n：longint])　変数をデクリメントする．xは順序型変数．nが指定されているときnずつ，指定されていないとき1ずつ減少する．

・Inc(var x[；n：longint])　変数をインクリメントする．xは順序型変数．nが指定されているときnずつ，指定されていないとき1ずつ増加する．

・b：= Odd(x：longint)　奇数ならtrue，偶数ならfalseを返す．

・y：= Pred(x)　引数xの前の要素を返す．結果の型は引数の型に等しい．

・y：= Succ(x)　引数xの次の要素を返す．結果の型は引数の型に等しい．

・L：= Round(x：real)　実数型の値を小数以下四捨五入し，整数型に変換する．xは実数型の式で，結果はlongint型になる．オーバーフローしたときはエラーとなる．

・L：= Trunc(x：real)　実数型の値を小数以下を切り捨て，整数型に変換する．xは実数型の式で，結果はlongint型になる．オーバーフローしたときはエラーとなる．

7.　文字列関係

・Delete(var s：string；index：integer；count：integer)　文字列sのindex番目からcount個の文字を削除する．

・Insert(source：string；s：string；index：integer)　文字列sourceを文字列sのindex番目の前の位置に挿入する．結果の文字数が255を越えると，越えた分は削除される．

・Str(x[：width[：decimals]]；var s：string)　数値を文字列に変換する．xは整数または実数型の式，width, decimalsは整数型の式である．

・Val(s：string；var v；var code：integer)　文字列sを数値に変換し変数vに代入する．結果のcodeが0でないときは，sのcode番目の文字でエラーが生じたことを意味する．vは整数型または実数型の変数．

・c：= Chr(x：byte)　文字コードに対する文字を返す．

・s：= Concat(s1[，….，sN])　文字列s1からsNまでを結合した文字列を返す．s：= s1+s2+…+sNに等しい．結果の文字列数が255を越えた場合，越えた分は削除される．

・s：= Copy(st：string；index：integer；count：integer)　文字列stのindex番目からcount個の文字列を返す．

・i：= Length(s：string)　文字列sの動的長さ，つまり現在の文字数を返す．結果は整数型である．i：= ord(st[0])に等しい．

・i：= Pos(substr, s：string)　文字列 s 中の文字列 substr がはじめにある位置を返す．ない場合は 0 を返す．

・i：= Ord(c)　文字 c のコードを整数値として返す．

・ch：= UpCase(c)　文字 c を大文字に変換する．'a'..'z' の範囲外にあるものは何の影響も受けない．

8. ポインタアドレス関係

・p：= Addr(v)　@演算子と同じ働きをする．v は変数名，手続き・関数名．アドレスをポインタ型で返す．

・w：= Cseg　現在のコードセグメントを返す．結果は word 型．

・w：= Dseg　現在のデータセグメントを返す．結果は word 型．

・w：= Ofs(x)　x で示される変数・手続き・関数のアドレスのオフセットを返す．結果は word 型．

・p：= Ptr(seg, ofs：word)　セグメント seg とオフセット ofs で表されるアドレスをポインタ型で返す．

・w：= Seg(x)　x で示される変数・手続き・関数のアドレスのセグメントを返す．結果は word 型．

・w：= SPtr　現在のスタックポインタを返す．結果は word 型．

・w：= Sseg　現在のスタックセグメントを返す．結果は word 型．

9. 動的変数関係

・Dispose(var p：pointer)　動的変数を処分し，それが占めていたメモリを開放する．p は New 手続きであらかじめ割り当てられたポインタ型変数．

・FreeMem(var p：pointer ; size：word)　与えられたサイズの動的変数を破棄する．getmem で確保した領域を開放する．

・GetMem(var p：pointer ; size：word)　指定されたサイズの動的変数を作り，領域のアドレスを動的変数に格納する．

・Mark(var p：pointer)　変数にヒープポインタの値を代入する．

・New(var p：pointer)　新しい動的変数を生成し，ポインタ変数をセットする．

・Release(var p：pointer)　ヒープポインタを引数指定の状態にセットする．

・L：= MaxAvail　ヒープ用領域の最大の未使用ブロックサイズを返す．結果は longint 型．

・L：= MemAvail　ヒープ用領域の未使用ブロックの合計を返す．結果は longint 型．

10. そ の 他

・FillChar(var x ; count：word ; ch：char)　x は任意の型である．x の先頭から count バイト個先のアドレストに ch の値を書き込む．ch は文字型またはバイト型の式．

・Move(var source, dest ; count：word)　指定された連続したバイト数 count をある領域 source から別の領域 dest へ転送する．source と dest は任意の型の変数である．

・i：= Hi(x)　引数 x の高位バイトを返す．x は integer または word 型の式で結果は byte 型となる．

・i：= Lo(x)　引数 x の低位バイトを返す．x は integer または word 型の式で結果は byte 型となる．

・w：= ParamCount　コマンド行でプログラムに渡されたパラメータ数を返す．空白とタブは区切りになる．結果は word 型．

・s：= ParamStr(index)　コマンド行の index 番目の文字列を返す．index = 0 のときは実行されているプログラムのパス・ファイル名を返す．

・w：= SizeOf(x)　変数または型名 x の占めているメモリ中のバイト数を返す．結果は word 型．

・w：＝Swap(x：word)　　引数の上位バイトと下位バイトの交換した値を返す．結果は word 型．

付録B　Crt ユニット

1.　定数，型，変数

・crt モード定数

VL8025	VL8020	VL4025	VL4020	BG8025	BG8020	BG4025	BG4020
0	1	2	3	4	5	6	7

・テキストカラー定数

Black	Blue	Green	Cyan	Red	Magenta	Yellow	White
$00	$20	$80	$A0	$40	$60	$C0	$E0

・テキスト属性定数

VerticalLine	NoVerticalLine	UnderLine	NoUnderLine	Reverse	NoReverse
$10	$00	$08	$00	$04	$00

Blink	NoBlink	Secret	NoSecret	BasicGraph	NoBasicGraph
$02	$00	$00	$01	$10	$00

・カーソル属性定数

DispCursor	NoDispCursor	BlinkCursor	NoBlinkCursor
$01	$00	$00	$02

・Crt 変数

CheckBreak：boolean　STOP キーをチェックする/しない

CheckEOF：boolean　ファイル終了文字を有効/無効

LastMode：word　TextMode が呼ばれるごとにビデオモードが記憶される．

TextAttr：byte　現在のテキスト属性

WindMin：word　ウィンドウの左上の座標，x＝上位バイト，y＝下位バイト

WindMax：word　ウィンドウの右下の座標，x＝上位バイト，y＝下位バイト

Kanji：boolean　漢字の取り扱い

DelayFactor：word　delay でプログラムが休止する時間を制御する．

2.　手続き・関数

AssignCrt(var f：text)　テキストファイル f を CRT(画面)と結び付ける．これにより，標準出力(入力)は高速に出力(入力)される．

Beep(Time：word)　ビープ音を発生させる．Sound で設定されている周波数を Time 回発生する．

ClrEol　カーソル位置よりその行の行末までの文字を消去する．

ClrScr　画面を消去し，カーソルを上左端に置く．

Delay(ms：word)　ms ミリ秒の遅延を行う．DelayFactor の値で調節する．

DelLine　カーソル行の消去し，スクロールアップを行う．

GotoXY (x, y：byte)　カーソルを (x, y) に移動する．ウィンドウの相対座標を用いる．

InsLine　カーソルのある場所に1行挿入する．

NormVideo　起動時のカーソル位置から読み取ったオリジナルのテキスト属性を選択する．

Sound (Frequency：word)　ビープ音の周波数を設定する．Frequency は周波数の逆数として使われる．

TextBank (bank：word)　画面ページを切り換える．Bank は0か1である．これを用いると2つのテキスト画面の表示を瞬時に切り換えることができる．

TextBlink (BlinkAttr：byte)　文字の点滅 (ブリンク) 属性を選択する．Blink (= 2) で点滅，NoBlink (= 0) で普通の表示．

TextColor (Color：byte)　文字の色属性を設定する．

TextCursor (CursorAttr：byte)　カーソル属性を設定する．

TextMode (Mode：word)　指定のテキストモードを選択する．

TextReverse (ReverseAttr：byte)　文字の反転 (リバース) 属性を設定する．

TextUnder (UnderAttr：byte)　文字の下線属性を設定する．

TextVertical (VerticalAttr：byte)　文字の縦罫線属性を設定する．

Window (x1, y1, x2, y2：byte)　画面上にテキストウィンドウを定義する．左上が (x1, y1)，右下は (x2, y2) である．

x：= KeyPressed　キーが押されていれば true，押されていなければ false を返す．

c：= ReadKey　キーボードから1文字読み込む．結果は文字型である．画面上には表示されない．入力がないときは，入力されるまで待つ．

i：= WhereX　カレントカーソルポジションの x 座標をウィンドウに対する相対位置で返す．結果は byte 型である．

i：= WhereY　カレントカーソルポジションの y 座標をウィンドウに対する相対位置で返す．結果は byte 型である．

付録C　Dosユニット

1.　定数，型，変数

・フラグ定数

FCarry, FParity, FAuxiliary, FZero, FSign, FOverflow

$1　$4　$10　$40　$80　$800

・ファイルモード定数

fmClosed, fmInput, fmOutput, fmInOut

$D7B0　$D7B1　$D7B2　$D7B3

・ファイルレコード型

```
type
  FileRec = record
              Handle, Mode, RecSize : word ;
              Private : array[1..26] of byte ;
              UserData : array[1..16] of byte ;
              Name : array[0..79] of byte ;
            end ;
  TextBuf = array[0..127] of char ;
  TextRec = record
              Handle, Mode, Bufsize, Private, BufPos, Bufend : word ;
              BufPtr : ^TextBuf ;
              OpenFunc, InOutFunc, FlushFunc, CloseFunc : pointegr ;
              UserData : array[1..16] of byte ;
              Name : array[0..79] of byte ;
            end ;
```

・ファイル属性定数

ReadOnly, Hidden, SysFile, VolumeID, Directory, Archive, AnyFile

$01　$02　$04　$08　$10　$20　$3F

・Registers 型

```
type
  Registers=record
              case integer of
              0 : (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags : word) ;
              1 : (AL, AH, BL, BH, CL, CH, DL, DH : byte) ;
            end ;
```

・DateTime型

```
type
  DateTime=record
    Year, Month, Day, Hour, Min, Sec : integer;
  end;
```

・SearchRec型

```
type
  SearchRec=record
    Fill : array[1..21] of byte;
    Attr : byte;
    Time : longint;
    Size : longint;
    Name : string[12];
  end;
```

・ファイル操作に関する文字列型

```
type
  DirStr=string[67];
  NameStr=string[8];
  ExtStr=string[4];
```

・DosError変数

```
var DosError : integer;
```

0：エラーなし	8：メモリ不足
2：ファイルが見つからない	10：無効な環境
5：アクセスできない	11：無効な書式
6：無効なハンドル	18：これ以上ファイルはない

2.　手続き・関数

・GetIntVec(IntNo : byte ; var vector : pointer)　　指定された割り込みベクタに格納されているアドレスを返す.

・Intr(IntNo : byte ; var Regs : Registers)　　指定されたソフトウェア割り込みを実行する.

・MsDos(var Regs : Registers)　　MS-DOSファンクションコールを実行する.

・SetIntVec(IntNo : byte ; Vector : pointer)　　指定された割り込みベクターに指定のアドレスをセットする.

・GetDate(var Year, Month, Day, DayOfWeek : word)　　現在の日付を返す.

・GetFTime(var f ; var Time : longint)　　ファイルに書き込みが行われた最後の日付と時刻を返す.

・GetTime(var Hour, Minute, Second, Sec100 : word)　　現在の時刻を返す.

・PackTime(var DT : DateTime ; var Time : longint)　　DateTime型レコードを，SetFTimeで用いられる4バイトパック形式の日付・時刻を含むlongint型に変換する.

・SetDate(Year, Month, Day : word)　　日付を設定する.

・SetFTime(var f ; Time : longint)　　ファイルが最後に更新された日付と時刻をセットする.

・SetTime(var Hour, Minute, Second, Sec100 : word)　　システムの時計に時刻をセットする．指定した時刻が有効でない場合は無視される.

・UnpackTime (Time : longint ; var DT : DateTime)　4バイトパック形式の日付・時刻を含む longint 型をパックされていない DateTime 型に変換する
・L : = DiskFree (Drive : word)　指定したディスクドライブの未使用領域のバイト数を返す．Drive = 0 がカレントドライブ，Drive = 1 がドライブ A，Drive = 2 がドライブ B……となる．結果は longint 型で，エラーのときは−1となる．
・L : = DiskSize (Drive : word)　指定したディスクドライブの総バイト数を返す．Drive = 0 がカレントドライブ，Drive = 1 がドライブ A，Drive = 2 がドライブ B……となる．結果は longint 型で，エラーのときは−1となる．
・FindFirst (Path : string ; Attr : word ; var S : SearchRec)　指定された(あるいはカレント)ディレクトリから指定されたファイル名と属性に一致する最初のエントリを捜す．
・FindNext (var s : SearchRec)　以前の FindFirst の呼び出しで指定されたファイル名と属性に一致する次のエントリを返す．
・Fsplit (Path : PathStr ; var Dir : DirStr ; Var Name : NameStr ; var Ext : ExtStr)
ファイル名を3つの部分に分解する．
・GetFAttr (var f ; var Attr : word)　ファイルの属性を返す．
・SetFAttr (var f ; var Attr : word)　ファイルの属性を設定する．
・s : = FExpand (path : Pathstr)　ファイル名を，完全なパス名を含むファイル名に拡張する．結果は Pathstr 型 (string [79])．
・s : = FSearch (Path : PathStr ; DirList : string)　指定された複数のディレクトリの中から指定されたファイルを返します．結果は Pathstr 型 (string [79])．
・Exec (Path, Cmdline : string)　指定されたプログラムを指定されたコマンドラインで実行する．
・Keep (ExitCode : word)　プログラムをメモリに常駐したまま終了させる．
・SwapVectors　割り込みベクターを入れ換える．
・W : = DosExitCode　サブプロセスの終了コードを返す．
・i : = EnvCount　DOS 環境にセットされている文字列の数を返す．結果は integer 型である．
・s : = EnvStr (index : integer)　指定された環境文字列を返す．
・s : = GetEnv (EnvVar : string)　指定された環境変数の値を返す．
・w : = DosVersion　DOS のバージョン番号を返す．結果は word 型で上位バイトに整数部，下位バイトに小数部が入る．
・GetCBreak (var Break : boolean)　DOS における CTRL-C チェックの状態を返す．
・GetVerify (var Verify : boolean)　DOS のベリファイフラグの状態を返す．
・SetCBreak (Break : boolean)　DOS における CTRL-C の状態を設定する．
・SetVerify (Verify : boolean)　DOS のベリファイフラグの状態をセットする．

付録 D　Graph ユニット

1.　定数，型，変数

エラーコード

grOk,　grNoInitGraph, grNotDetected, grFileNotFound,
0　−1　−2　−3

grInvalidDriver, grNoLoadMem, grNoScanMen, grNoFloodMem,
−4　−5　−6　−7

grFontNotFound, grNoFontMem, grInvalidMode, grError,
−8　−9　−10　−11

grIOerror, grInvalidFont, grInvalidFontNum, grInvalidDeviceNum,
−12　−13　−14　−15

grInvalidPrintInit, grPrintNotLinked, grInvalidVersion
−16　−17　−18

グラフィックドライバー

CurrentDriver, Detect
−128　0

CGA, MCGA, EGA, EGA64, EGAMono, IBM8514, HercMono, ATT400
1　2　3　4　5　6　7　8

VGA, PC3270, PC98, PC98GRCG, PC98EGC, FMRL, FMRM, FMRH
9　10　11　12　13　14　15　16

グラフィックモード

FMRC1, FMRC8, FMRC16
0　1　2

最大色数

MaxColors
15

色コード

Black, Blue, Green, Cyan, Red, Magenta, DarkYellow,
0　1　2　3　4　5　6

Brown, LightGray, DarkGray, LightBlue, LightGreen,

6 7 8 9 10

LightCyan, LightRed, LightMagenta, Yellow, White,

11 12 13 14 15

線の種類

SolidLn, DottedLn, CenterLn, DashedLn, UserBitLn

0 1 2 3 4

線の幅

NormWidth, ThickWidth

1 3

テキスタイル定数

DefaultFont, TriplexFont, SmallFont, SansSerifFont, GothicFont

0 1 2 3 4

HorizDir, VertDir, UserCharSize

0 1 0

クリッピング定数

ClipOn, ClipOff

true false

3 次元バー定数

TopOn, TopOff

true false

塗りつぶしパターン定数

EmptyFill, SolidFill, LineFill, LtSlashFill,

0 1 2 3

SlashFill, BkSlashFill, LtBkSlashFill, HatchFill,

4 5 6 7

XHatchFill, InterleaveFill, WideDotFill,

8 9 10

CloseDotFill, UserFill

11 12

イメージ出力定数

NormalPut, CopyPut, XORPut, OrPut, AndPut, NotPut

0 0 1 2 3 4

テキスト位置合わせ定数

LeftText, CenterText, RightText
  0　　1　　2

BottomText, CenterText, TopText
  0　　1　　2

・型

```
type
    PalletType = record
      Size   : byte ;
      Colors : array [0..MaxColors] of shortint ;
    end ;

type
    LineSettingsType = record
      LineStyle : word ;
      Pattern   : word ;
      Thickness : word ;
    end ;

type
    TextSettingsType = record
      Font      : word ;
      Direction : word ;
      CharSize  : word ;
      Horiz     : word ;
      Vert      : word ;
    end ;

type
    FillSettingsType = record
      Pattern : word ;
      Color   : word ;
    end ;

type
    FillPatternType = array [1..8] of byte ;

type
    ViewPortType = record
      x1, y1, x2, y2 : integer ;
      clip           : boolean ;
```

```
        end ;

type
        ArcCoordsType = record
          x, y           : integer ;
          Xstart, Ystart : integer ;
          Xend, Yend     : integer ;
        end ;
```

・変数

```
var
        GraphGetMemPtr  : pointer ;
        GraphFreeMemPtr : pointer ;
```

2. 手続き・関数

- Arc (x, y : integer ; StAngle, EndAngle, Radius : word)　(x, y) を中心として始角から終角まで円弧を描く．
- Bar (x1, y1, x2, y2 : integer)　(x1, y1) と (x2, y2) を頂点とする長方形を描いて内部を塗りつぶす．
- Bar3D (x1, y1, x2, y2 : integer ; Depth : word ; Top : boolean)　(x1, y1) と (x2, y2) を頂点とする長方形を 3 次元的に描いて内部を塗りつぶす．
- Circle (x, y : integer ; Radius : word)　中心 (x, y)，半径 r の円を描く．
- ClearDevice　グラフィックス画面を消去する．
- ClearViewPort　カレントビューポートを消去する．
- CloseGraph　グラフィックスの終了．
- DetectGraph (var GraphDriver, GraphMode : integer)　ハードウエアをチェックしてドライブとモードを決定する．DetectGraph を Detect と省略することができる．
- DrawPoly (NumPoints : word ; var PolyPoints)　多角形の輪郭を描く．
- Ellips (x, y : integer ; StAngle, EndAngle : word ; XRadius, YRadius : word)
  中心 (x, y)，始角から終角までの楕円弧を描く．
- FillEllips (x, y : integer ; XRadius, YRadius : word)　楕円を描いて内部を塗りつぶす．
- FillPoly (NumPoints : word ; var PolyPoints)　多角形を描いて内部を塗りつぶす．
- FloodFill (x, y : integer ; Border : word)　閉じた領域をカレントフィルパターンを用いて塗りつぶす．
- GetArcCoords (var ArcCoords : ArcCoordsType)　直前の Arc コマンドの座標を得る．
- GetAspectRatio (var Xasp, Yasp : word)　アスペクト比を計算するためのグラフィック画面の実行分解能を得る．
- GetDefaultPalette (var Palette : PaletteType)　デフォルトのカラーパレットに関する情報を得る．
- GetFillPattern (var FillPattern : FillPatternType)　SetFillPattern でセットされたフィルパターンを返す．
- GetFillSettings (var FillInfo : FillSettingsType)　SetFillStyle あるいは SetFillPattern でセットされたカレントフィルパターンとカラーを返す．
- GetImage (x1, y1, x2, y2 : integer ; var BitMap)　指定された領域のビットイメージをバッファにセーブする．

・GetLineSettings (var LineInfo : LineSettingsType)　SetLineStyle によって設定されているカレントラインスタイル，ラインパターン，ライン幅を返す．
・GetModeRange (graphDriver : integer ; var LoMode, HiMode : integer)　最大のグラフィックスモードを返す．
・GetPalette (var Palette : PaletteType)　カレントパレットおよびそのサイズを返す．
・GetTextSettings (var TextInfo : TextSettingsType)　SetTextStyle および SetTextJustify によりセットされたカレントテキストフォント，方向，サイズ，位置合わせを返す．
・GetViewSettings (var ViewPort : ViewPortType)　SetViewPort によりセットされたカレントビューポートとクリッピングの設定を得る．
・GraphDefaults　グラフィックスの設定をリセットする．
・InitGraph (var GraphDriver, GraphMode : integer ; DriverPath : string)　グラフィックシステムを初期化し，ハードウェア情報をグラフィックモードに返す．
・Line (x1, y1, x2, y2 : integer)　(x1, y1) から (x2, y2) へ直線を引く．
・LineRel (dx, dy : integer)　カレントポインタから指定の相対位置にある点まで直線を引く．
・LineTo (x, y : integer)　カレントポインタから (x, y) まで直線を引く．
・MoveRel (dx, dy : integer)　カレントポインタを指定の相対位置にある点まで移動する．
・MoveTo (x, y : integer)　カレントポインタを (x, y) に移動する．
・OutText (TextString : string)　出力デバイスのカレントポインタの位置に文字列を書き出す．
・OutTextXY (x, y : integer ; TextString : string)　出力デバイスの (x, y) の位置に文字列を書き出す．
・PieSlice (x, y : integer ; StAngle, EndAngle, Radius : word)　(x, y) を中心に始角から終角まで扇形を描いて塗りつぶす．
・PieSliceXY (x, y : integer ; StAngle, EndAngle, XRadius, YRadius : word)　(x, y) を中心に始角から終角まで楕円の扇形を描いて塗りつぶす．
・PutImage (x, y : integer ; var BitMap, BitBlt : word)　ビットイメージを画面上に出力する．
・PutPixel (x, y : integer ; Color : word)　(x, y) に点をプロットする．
・RectAngle (x1, y1, x2, y2 : integer)　カレントラインスタイルとカラーを使って長方形を描く．
・RegisterBGIdriver (driver : pointer)　ユーザがロードした BGI ドライバ，あるいはリンクされたドライバをグラフィックシステムに登録する．
・Sector (x, y : integer ; StAngle, EndAngle, XRadius, YRadius : word)　(x, y) を中心に始角から終角まで楕円の扇形を描いて塗りつぶす．
・SetActivePage (page : word)　グラフィックス出力の対象となるアクティブページをセットする．
・SetAllPalette (var Palette)　すべてのパレットカラーを指定されたように変更する．
・SetAspectRatio (var Xasp, Yasp : word)　デフォルトのアスペクト比を変更する．
・SetBkColor (Color : word)　パレットを使用してカレントバックグランドカラーをセットする．
・SetColor (Color : word)　パレットを使用してカレントドロウカラーをセットする．
・SetFillPattern (pattern : FillPatternType ; Color : word)　ユーザ定義のフィルパターンを選択する．
・SetFillStyle (pattern, Color : word)　フィルパターンとカラーをセットする．
・SetGraphBufSize (BufSize : word)　スキャンフィルとフラッドフィルのために使用されるバッファサイズを変更する．
・SetGraphMode (Mode : integer)　システムをグラフィックスモードにセットし，画面をクリアする．
・SetLineStyle (LineStyle, Pattern, Thickness : word)　カレントライン幅，スタイルをセットする．
・SetPalette (ColorNum : word ; Color : shortint)　パレットカラーを変更する．

・SetRGBPalette (ColorNum, RedValue, GreenValue, BlueValue : integer)　RGB パレットを変更する.
・SetTextJustify (Horiz, Vert : word)　OutText, OutTextXY によって使用されるテキストの位置合わせを表す値をセットする.
・SetTextStyle (Font, Direction, CharSize : word)　カレントテキストフォント，スタイル，文字倍率をセットする.
・SetUserCharSize (MultX, DivX, MultY, DivY : word)　ストロークフォントの文字の幅と高さを変更する.
・SetViewPort (x1, y1, x2, y2 : word ; clip : boolean)　カレント出力ビューポート，あるいはウィンドウを設定する.
・SetVisualPage (page : word)　ビジュアルグラフィックスページ番号をセットする.
・SetWriteMode (WriteMode : integer)　グラフィック出力の書き込みモードを設定する.

・w : = GetBkColor　カレントバックグランドカラーへのインデックスを返す.
・w : = GetColor　直前の SetColor の値を返す.
・s : = GetDriverName　カレントグラフィックスドライバの名前を含む文字列を返す.
・i : = GetGraphMode　グラフィックモードを返す.
・w : = GetMaxColor　SetColor に渡すことのできる最大のカラー値を返す.
・i : = GetMaxMode　カレントグラフィックスドライバで使用できる最大のモード番号を返す.
・w : = GetMaxX　カレントグラフィックスドライバおよびモードにおける右端の座標値を返す.
・w : = GetMaxY　カレントグラフィックスドライバおよびモードにおける下端の座標値を返す.
・s : = GetModeName (GraphMode : integer)　グラフィックモードの名前を含む文字列を返す.
・i : = GetPaletteSize)　パレットカラー参照テーブルのサイズを返す.
・w : = GetPixel (x, y : integer)　座標におけるピクセルの値を返す.
・i : = GetX　カレントポインタの x 座標を返す.
・i : = GetY　カレントポインタの y 座標を返す.
・s : = GraphErrorMsg (ErrorCode : integer)　指定されたエラーコードに対するエラーメッセージ文字列を返す.
・i : = GraphResult　直前のグラフィックス操作に対するエラーコードを返す.
・i : = ImageSize (x1, y1, x2, y2 : integer)　画面上の長方形の領域の格納に必要なバイト数を返す.
・w : = InstallUserDriver (DriverFileName : string ; AutoDetectPtr : pointer)
　他のグラフィックスドライバを BGI デバイスドライバテーブルに登録する.
・i : = InstallUserFonts (FontFileName : string)　新しいフォントをインストールする.
・i : = RegisterBGIfont (font : pointer)　ユーザがロードした BGI フォント，あるいはリンクされたフォントをグラフィックシステムに登録する.
・w : = TextHeight (TextString : string)　文字列の高さをピクセル単位で返す.
・w : = TextWidth (TextString : string)　文字列の幅をピクセル単位で返す.

付録E　コンパイラ・コマンド

注釈のうちの{$ 制御文字，制御文字，…，制御文字}

制御文字	意　味
A+	データのアラインメントをワード(標準値)
A−	データのアラインメントをバイト
B+	論理式の評価を完全にする(標準値)
B−	論理式の評価をショートサーキットにする
D+	デバッグ情報を生成する(標準値)
D−	デバッグ情報を生成しない
E+	数値演算プロセッサエミュレーション(標準値)
E−	数値演算プロセッサのエミュレーションをしない
F+	FAR コールの強制をする
F−	FAR コールの強制をしない(標準値)
I+	I/O のエラーチェックを行なう(標準値)
I−	I/O のエラーチェックを行なわない
L+	ローカルシンボル情報の生成(標準値)
L−	ローカルシンボル情報を生成しない
N+	数値演算プロセッサコードの生成
N−	数値演算プロセッサコードを生成しない(標準値)
O+	オーバーレイコードの生成
O−	オーバーレイコードを生成しない(標準値)
R+	添字の範囲のチェックを行なう
R−	添字の範囲のチェックを行なわない(標準値)
S+	スタックのチェックを行なう(標準値)
S−	スタックのチェックを行なわない
V+	変数引数の型のチェックを行なう(標準値)
V−	変数引数の型のチェックを行なわない

I ファイル名　指定されたファイルを読み込んでコンパイルする

L ファイル名　オブジェクトファイルのリンク

M stack, heapmin, heapman　スタック領域のサイズ指定，ヒープ領域の最小値，最大値の指定

O ユニット名　ユニットをオーバーレイにする

和文索引

英 文 索 引

著者略歴

永野(ながの)三郎(さぶろう)

1963 年　東京大学工学部航空学科卒業
1968 年　同大学院修了，工学博士
現　在　東京大学教授(教養学部基礎科学科第二)

長島(ながしま)　忍(しのぶ)

1979 年　東京大学工学部精密機械工学科卒業
1985 年　工学博士
現　在　東京大学講師(教養学部情報・図学教室)

吉村(よしむら)　伸(しん)

1983 年　東京大学理学部生物化学科卒業
1985 年　同大学院(化学)修了
現　在　東京大学助手(教養学部化学教室)

Pascal プログラミング［第 2 版］
——TURBO Pascal による基礎と応用

1987 年 4 月 30 日　初　版第 1 刷
1992 年 3 月 10 日　第 2 版第 1 刷

［検印廃止］

著　者　永野三郎・長島 忍・吉村 伸

発行所　財団法人 東京大学出版会

代 表 者　養老孟司
113 東京都文京区本郷 7-3-1 東大構内
電話03(3811)8814・振替東京 6-59964

印　刷　株式会社平文社
製　本　有限会社島崎製本所

ISBN 4-13-062133-5　Printed in Japan